Algoritmos y Estructuras de Datos III

Primer Cuatrimestre de 2017

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Trabajo Práctico 3

Integrante	LU	Correo electrónico
Nicolás Ippolito	724/14	ns_ippolito@hotmail.com
Emiliano Galimberti	109/15	emigali@hotmail.com
Gregorio Freidin	433/15	gregorio.freidin@gmail.com
Pedro Mattenet Riva	428/15	peter_matt@hotmail.com

Reservado para la catedra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

Contents

1 Utilidades en la vida Real **3**
1.1 El Grupo Social Más Famoso . . . 3
1.2 El Mercado Más Influyente . . . 3
1.3 El Grupo Más Toxico . . . 3
1.4 BANE ATACA DE NUEVO! . . . 3
1.4.1 Modificación BANE . . . 3

2 Algoritmo Exacto **4**
2.1 Pseudocódigo y complejidad teórica . . . 4
2.2 Experimentación . . . 6

3 Heurística Golosa **7**
3.1 MFCGM . . . 7
3.1.1 Instancias no óptimas . . . 7
3.1.2 PseudoCódigo y Complejidad teórica . . . 8
3.2 Máxima Clique Mayor Frontera . . . 10
3.2.1 Instancias no optimas . . . 10
3.2.2 PseudoCódigo y Complejidad teórica . . . 11
3.3 Experimentación . . . 12
3.3.1 Tiempo de cómputo . . . 12
3.3.2 Heurística 1 vs Heurística 2 vs Algoritmo Exacto . . . 12
3.3.3 Experimentación general . . . 14

4 Heurística Búsqueda Local **15**
4.1 Vecindad Lineal . . . 15
4.2 Vecindad Cuadrática . . . 15
4.3 Implementación . . . 16
4.3.1 Vecindad Lineal . . . 16
4.3.2 Vecindad Cuadrática . . . 19
4.4 Experimentación . . . 19
4.4.1 Experimento 2 - Comportamiento Temporal . . . 24

5 Meta-Heurística GRASP **27**
5.1 Heuristica Golosa . . . 27
5.2 PseudoCódigo . . . 27
5.3 Experimentación . . . 29
5.3.1 Cantidad Iteraciones . . . 29
5.3.2 Parámetro α óptimo . . . 30
5.3.3 Iteraciones Permitidas vs Realizadas . . . 30
5.3.4 Experimentación general . . . 31

6 Resultados finales **34**
6.1 Comparación Exacto vs Heurísticas . . . 34
6.2 Comparando Resultados . . . 35
6.3 Análisis de tiempos . . . 38

1 Utilidades en la vida Real

1.1 El Grupo Social Más Famoso

Modelando a las personas como nodos, si una persona comparte un grupo con otra, sus respectivos nodos están conectados por una arista. Si la persona está conectada a muchas otras, o bien que el grado de su nodo es alto, decimos que es $famosa$. Se busca encontrar el grupo social más famoso de todos, esto es, el grupo cuyos integrantes están conectados a más personas fuera de éste. Notar que un grupo puede estar formado por sólo una o dos personas

1.2 El Mercado Más Influyente

Representando comercios como nodos, si un comercio tiene negocios con otro, entonces sus nodos estarán unidos por una arista (no importa quién vende o quién compra, las aristas no tienen dirección). Se denomina *mercado* al subconjunto de comercios entre los cuales todos hacen negocios entre sí (sería nuestra clique). Un comercio es $influyente$ si tiene muchos negocios con otros comercios. Se busca entonces, el Mercado Más Influyente. Esto es, el subconjunto de comercios (que son mercado) que más negocios poseen con los comercios de afuera del mercado.

1.3 El Grupo Más Toxico

En un conocido juego MOBA online, la compañía creadora del juego (RITO GAMES) nos pidió ayuda. Se dice que un jugador es **tóxico** cuando insulta, agrede o discrimina a otro jugador. Generalmente, como los jugadores son de poca edad madurativa, al ser *entoxicados* por un jugador devuelven los insultos, generando un enlace entre los dos nodos, de manera tal que el jugador del nodo A y el jugador del nodo B están unidos por una arista sin dirección cuando ambos son tóxicos entre si. Se desea encontrar al subconjunto de jugadores, todos tóxicos entre si, que insulten, agredan o discriminen a más jugadores de afuera del grupo.

1.4 BANE ATACA DE NUEVO!

Así como dice el título, Bane ataca de nuevo! El joven villano de la saga DC nacido en la República Caribeña de Santa Prisca se quedó frustrado por su intento fallido de destruir Gotham City ya atrás en el 2012.

Con una perfecta paciencia estuvo esperando el momento indicado para anunciar sus intenciones de regreso, y tras la muerte del histórico Batman (W.W.Anderson), más bien conocido como Adam West, sabe bien que ya nadie podrá detenerlo.

Sin embargo esta vuelta no logró conseguir la Visa para volver a los Estados Unidos. Por lo que sin mucha sorpresa, al ser conocida la historia de enemistad de este personaje con las pastas, su siguiente objetivo para aterrorizar ha pasado a ser la bella Italia.*****

Así bien la noticia en particular llegó a la vieja Venecia, cuyas autoridades sin más preámbulo, como política de seguridad, decidieron reacomodar su demografía de forma que si Bane llega a atacarla, le cueste mucho cubrir todos los caminos de salida.

Prontamente salieron condiciones de cómo se debería llevar esto a cabo esto, principalmente una. Que todas las islas a las que se muden estén interconectadas entre sí. Se quiere que la economía funcione mas allá de todo mal. Y les pareció que esta idea de interconexion entre ella sería la más simple para lograrlo.

Rápidamente un joven despierto les aviso que este problema no tiene una solución exacta que se calcule de forma polinomial. Por lo que se abrió la búsqueda de quien les pueda aportar la mejor heurística para resolver su problema de reacomodamiento demográfico. Y la recompensa será MIL pizzas!

1.4.1 Modificación BANE

Realizamos una modificación a *Bane ataca de nuevo!* modificado desde el *****

Bane puede atacar todas las ciudades de Italia que quiera, pero sabe que debe intentar controlar su maldad pues mientras más ciudades ataque, más probabilidades tiene de fracasar en su cometido. Por otro lado, tiene una extraña obsesión con derribar puentes. Afirma que es porque le gusta que la gente víctima de su ataque quede encerrada (pero existe el rumor de que es una especie de fetiche sexual).

Luego, lo que busca Bane es atacar un conjunto de ciudades que estén conectadas entre sí, tales que la suma de todos sus puentes que las conectan con otras ciudades que no están en ese conjunto sea máxima. De esta forma, derribará muchísimos puentes y la gente se quedará encerrada entre ciudades que estarán controladas por él.
Si consigue esto, podrá llegar al pico de su maldad y logrará la iluminación divina.

Abstrayéndonos de lo descripto, lo que se busca es encontrar la clique de frontera máxima (siendo nodos las ciudades y aristas los puentes).

2 Algoritmo Exacto

Para resolver el problema descripto de forma exacta se resolvió utilizar la técnica de backtracking.

Teniendo los nodos numerados del 1 al n, partimos desde el nodo *1* ($i = 1$) y en cada paso:

- Utilizamos el nodo i
- No utilizamos el nodo i

De esta forma se arma un árbol de recursión que en cada hoja contiene una combinación específica de nodos utilizados y nodos no utilizados. Notar que si no hubiesen podas, las hojas del último nivel serían todas las combinaciones posibles (2^n).

Se implementó una poda para reducir el tiempo de cómputo. La misma consiste en determinar en tiempo lineal si el nodo i forma una clique con los anteriores de la rama correspondiente (en otras palabras, que sea adyacente a todos). Si no lo hace, entonces nos olvidamos de la rama recursiva que consiste en utilizar el nodo i.
De esta forma, al llegar a cada hoja del último nivel calcularemos su frontera (pues ya sabemos que es una clique) y nos quedaremos con la mayor de éstas -o sea: de las combinaciones de nodos que resultaron ser clique, aquella con la frontera más grande-.

2.1 Pseudocódigo y complejidad teórica

La idea del *Algoritmo 1* es crear todo lo necesario para que el algoritmo recursivo (*CliqueMaxFrontAux*) resuelva el problema.

Se decidió utilizar una matriz de adyacencia para representar al grafo.

Algorithm 1 CliqueMaxFront

```
procedure CliqueMaxFront(Graph grafo)
    n ← cantidad de nodos del grafo
    cliqueRes ← conjunto vacío
    cliqueAux ← conjunto vacío
    res ← 0
    i ← 0
    CliqueMaxFrontAux(grafo, n, i, cliqueAux, res, cliqueRes)
    tamClique ← #(cliqueRes)
    return (res, tamClique)
```

El conjunto *cliqueRes* contendrá luego del algoritmo que aplica el backtracking -*CliqueMaxFrontAux*- los nodos que pertenecerán a la clique de máxima frontera, y *cliqueAux* es un conjunto de nodos auxiliar que servirá para guardar los nodos utilizados en cada rama. Recordar que estamos recorriendo todas las combinaciones posibles de nodos que son cliques y definiendo su frontera.

Algorithm 2 CliqueMaxFrontAux

```
procedure CliqueMaxFrontAux(Graph grafo, int n, int ite, Conjunto<int> cliqueAux, int res, Conjunto<int> cliqueRes)
    if ite = n then
        cant ← CalculoFrontera(grafo, cliqueAux)
        if cant > res then                          ▷ Si la frontera de esta rama era mejor, la actualizo
            res = cant
            cliqueRes = cliqueAux

    if ite < n then
        if NoFormaClique(grafo, ite, cliqueAux) then      ▷ Si el nodo no forma una clique con los anteriores
            CliqueMaxFrontAux(grafo, n, ite + 1, cliqueAux, res, cliqueRes)      ▷ Solo rama de no usar nodo
        else
            CliqueMaxFrontAux(grafo, n, ite + 1, cliqueAux, res, cliqueRes)      ▷ Rama de no usar nodo
            Agregar(ite, cliqueAux)
            CliqueMaxFrontAux(grafo, n, ite + 1, cliqueAux, res, cliqueRes)      ▷ Rama de usar nodo
```

En el *Algoritmo 2* utilizamos algunas funciones:

- **calculoFrontera**. La frontera de una clique es igual a $\sum_{v \in clique}(d(v) - (|clique| - 1))$ pues la cantidad que cada nodo aporta a la frontera son todos sus vecinos, menos aquellos que pertenecen a la clique (|clique| - 1). Dado que utilizamos una matriz de adyacencia para representar a nuestro grafo, necesariamente en algunos casos (como en grafos completos) tendremos que recorrer toda la matriz para obtener el grado de cada nodo. Luego, la función tiene complejidad $O(n^2)$.
- **NoFormaClique**. Chequea si el nodo en cuestión es adyacente a todos los de la clique. Esto lo hacemos en complejidad lineal, pues tendremos representado a la clique como un arreglo de booleanos (donde a[i] = 1 si el nodo i está en la clique) lo que nos permite saber si cierto nodo pertenece a la clique en tiempo constante. Luego, sólo tendremos que recorrer la matriz de adyacencia que corresponde al nodo para saber si forma o no una clique.

Analizando mejor el *Algoritmo 2* vemos que en el peor caso -grafo completo, donde todo par de nodos es adyacente- en cada paso hago dos llamadas recursivas (usar y no usar el nodo) por lo que en total haría 2^n llamadas recursivas, cada una de ellas de complejidad lineal (por la función *NoFormaClique*). Al llegar a una hoja realizo una operación cuadrática (calcular su frontera). Luego, al llegar a cada una de las 2^n hojas voy a tener tener siempre una operación de complejidad cuadrática.

Finalmente, la complejidad es $O(2^n * n + 2^n * n^2) = O(2^n * n^2)$.

2.2 Experimentación

Los experimentos fueron realizados en una computadora con un procesador I5 4440 y 8GB de memoria RAM. La idea de la misma es analizar el comportamiento del algoritmo exacto con respecto al tiempo de cómputo para ver que efectivamente sería imposible su uso para instancias grandes. En particular, interesa ver el caso en el cual la poda no funciona (grafos completos).

Sea V la cantidad de nodos, en la experimentación se generaron los siguientes casos o familias de grafos:

- **T1**: Grafos que tienen V aristas.
- **T2**: Tienen al rededor de $V * \frac{V-1}{4}$ aristas. En otras palabras, tienen aproximadamente la mitad de las aristas que tendría un grafo completo.
- **T3**: Son grafos completos. Luego, en la matriz de adyacencia de esta familia de grafos G[i][j] = 0 sii i = j.

En las familias *T1* y *T2*, para determinar las adyacencias entre nodos se utilizó la librería random de C++ (siempre corrigiendo el caso en el cual estás buscando una adyacencia para el nodo i y random te devuelve i). En consecuencia, se aleatorizan las posibles cliqués de cada uno de éstos y, en particular, se hace difícil dar un número exacto de aristas en el caso *T2* -por lo que para describirlo se especificó que tienen "al rededor" de cierta cantidad de aristas-.

Para cada caso se crearon 40 instancias con distinta cantidad de nodos, desde 2 hasta 24 (se frenó en ese número ya que los grafos completos de 25 nodos tenían un tiempo de cómputo muy grande).

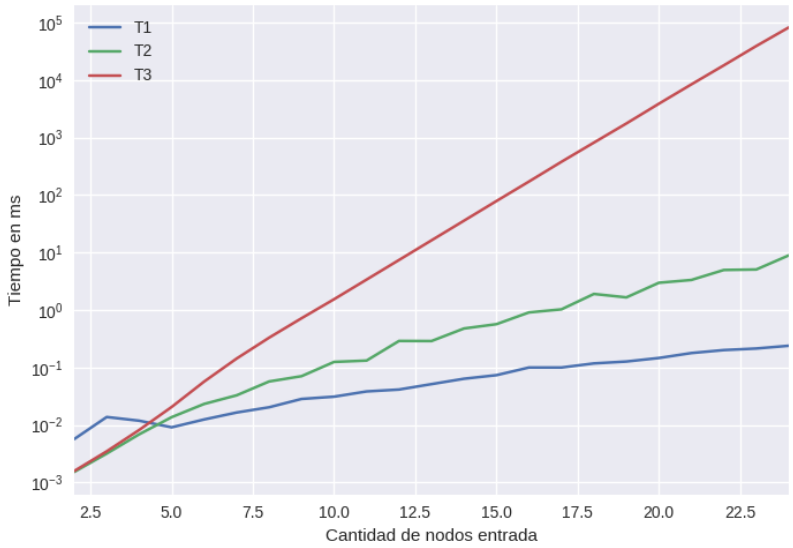


Figure 1: Tiempos con respecto a la cantidad de nodos de la entrada - escala logarítmica

Analizando la **Figura 1** vemos lo siguiente:

- *Los tiempos de cómputo del caso T3 son mucho más grandes que el resto.* Esto es completamente lógico y se debe a que para esta familia de grafos la poda no sirve ya que los mismos son completos. Luego, en cada rama recursiva se va a llegar al fondo del árbol (excepto en una: la combinación que corresponde a no utilizar ningún nodo) por lo que habrá $2^n - 1$ hojas en el último nivel donde cada una tendrá que pagar el costo n^2 de calcular su frontera.
- *Los tiempos de cómputo del caso T1 son muy pequeños en relación a los otros.* Tiene sentido ya que al haber pocas aristas la poda sería muy eficiente.

3 Heurística Golosa

En esta sección se plantean dos heurísticas golosas que tienen menor complejidad temporal que el algoritmo exacto, pero que no obtienen necesariamente un resultado óptimo.

3.1 MFCGM

La heurística *MFCGM* -máxima frontera cercana al grado máximo- consiste en lo siguiente:

- Buscamos el nodo de grado máximo *nodoGM* (o uno de ellos si fuesen varios). Notar que ésta ya es una posible clique de máxima frontera. Luego, lo agregamos a un conjunto que representará una clique.
- De todos los vecinos del último nodo agregado (*nodoGM* al comienzo) que forman una clique con todos los anteriores del conjunto (al principio son todos los vecinos de *nodoGM* pues 2 nodos adyacentes siempre forman una clique), nos quedamos con el que haga que esa clique tenga la máxima frontera entre todas las posibles (con esto nos referimos a las posibles agregándole un nodo adyacente al último agregado) y lo agregamos al conjunto. Luego, ahora la clique tendrá un nodo más y repetiremos el proceso hasta que no haya vecino del último nodo agregado que forme una clique con todos los anteriores.
- Finalmente, nos quedamos con la clique de máxima frontera entre todas las mejores que encontramos según cada cantidad de nodos (pues al agregar un nodo no necesariamente encontrás una frontera mejor).

Al finalizar estos pasos tengo una solución que se generó de forma golosa con un algoritmo de complejidad polinomial (se analizará más adelante).
No necesariamente será óptima, lo que también quedará claro en la sección **3.1.1**.

3.1.1 Instancias no óptimas

Vamos a presentar un tipo de grafos en el cual la heurística *MFCGM* no devuelve la solución óptima. Por supuesto que puede haber instancias menos óptimas que ésta, la idea fue construir una específica en la que la diferencia entre la solución verdadera y la solución de la heurística vaya incrementando a medida que crece la cantidad de nodos.

La familia de grafos que analizaremos tiene las siguientes características:

- Tiene dos componentes conexas.
- Una de las dos componentes conexas es una estrella (un nodo de grado de grado d adyacente a d nodos de grado 1).
- La otra componente conexa tiene dos nodos v y u adyacentes entre sí, donde además cada uno de ellos tiene otros $d-2$ nodos adyacentes de grado 1.

A este tipo de grafos los llamaremos **Estrella no óptima**. Abajo ilustramos un ejemplo de una instancia con 14 nodos.

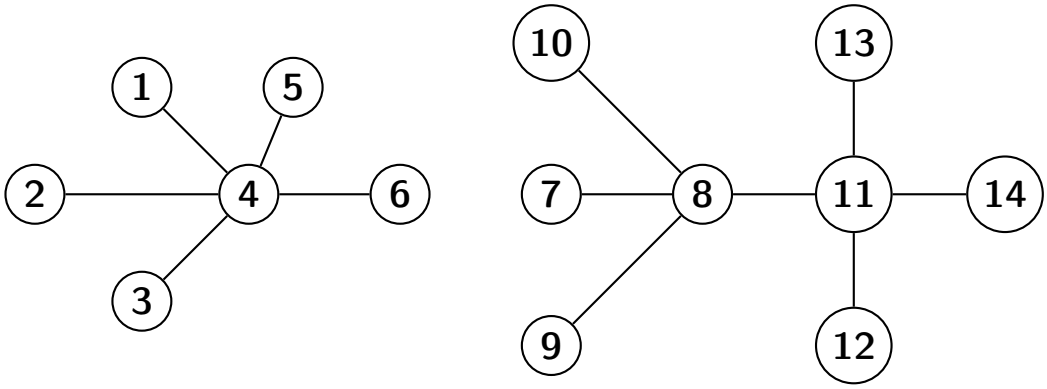


Figure 2: Instancia de 'Estrella no óptima' con 14 nodos

Analizando la **Figura 3** se ve que nuestra heurística se quedaría con la componente conexa de la izquierda (pues aquí está el nodo con mayor grado) y finalmente el algoritmo devolvería como resultado la clique de frontera máxima que está en esa componente -el conjunto de nodos [4] cuya frontera es 5-. Por otro lado, vemos que la clique de frontera

máxima en realidad está en la componente de la derecha y es el conjunto de nodos [8, 11] que tiene frontera 6. Luego, la heurística NO devuelve el resultado óptimo.

A su vez, vemos que partiendo de $n = 14$, a medida que aumentamos la cantidad de nodos del grafo la heurística devuelve cada vez soluciones más alejadas de la óptima. Sean *v, u* los nodos 8 y 11 respectivamente, y sea x el nodo 4:

- Si agregamos un nodo, lo ponemos como adyacente de cualquier nodo *v'* adyacente a v (distinto de u).
- Si agregamos dos nodos, sean *v'* y *u'* dos nodos adyacentes a v y u respectivamente (y ambos distintos de v y u), uno de los dos nodos agregados será adyacente a *v'* y el otro será adyacente a *u'*.
- Si agregamos tres nodos, uno de ellos será adyacente a x, otro adyacente a v y el restante adyacente a u. De esta forma, ahora la frontera de la clique [x] aumenta en 1, mientras que la frontera de la clique [v, u] aumenta en 2, por lo que la solución devuelta por la heurística se aleja aún más de la óptima.

Así, partiendo de una instancia que tenga una cantidad de nodos n tal que $n \equiv 2 \pmod 3$ y $n \geq 14$, se puede ir construyendo instancias más y más grandes (con respecto a la cantidad de nodos) que vayan siendo iguales en cuanto a la lejanía del resultado óptimo (si añado 1 o 2 nodos) o peores si añado más.

Por último observamos que si la cantidad de nodos es menor a 14, la familia de grafos *Estrella no óptima* no garantiza que la solución de nuestra heurística NO sea óptima, ya que con menos nodos no existe el caso en el cual el nodo x de la componente conexa estrella sea el único nodo de grado máximo y a la vez no sea la clique de frontera máxima (o una de ellas).

3.1.2 PseudoCódigo y Complejidad teórica

Decidimos representar al grafo como dos enteros (n y m que representan la cantidad de nodos y de aristas respectivamente) y una lista de listas (que representa la lista de adyacencias del grafo).

A continuación presentaremos el pseudocódigo del algoritmo y justificaremos su complejidad teórica.

Algorithm 3 MFCGMAux

```
1:  procedure MFCGMAux(Graph grafo)
2:      nodoGM ← nodoGradoMáximo(grafo)                          ▷ O(n)
3:      MejorFrontera ← Frontera(nodoGM)                         ▷ O(1)
4:      ultNodoAgregado ← nodoGM                                 ▷ O(1)
5:      Clique ← conjunto de nodos vacío                         ▷ O(1)
6:      Agregar(nodoGM, Clique)                                  ▷ O(1)
7:
8:      while Hay algún vecino de ultNodoAgregado que es adyacente a todos los elementos de Clique do
9:          for i in Vecinos(ultNodoAgregado) do                 ▷ O(∑_{j∈vecinos(i)} d(j))
10:             if i es vecino de todos los nodos de Clique then ▷ O(d(i))
11:                 CalcularFrontera(Clique ∪ i)                 ▷ O(1)
12:         PosibleMejorFrontera ← Mejor frontera de todas las que calculé en el for   ▷ O(1)
13:         k ← último nodo agregado a PosibleMejorFrontera                            ▷ O(1)
14:         Agregar(k, Clique)                                   ▷ O(1)
15:         if Frontera(Clique) > MejorFrontera then             ▷ O(1)
16:             MejorFrontera = Frontera(Clique)                 ▷ O(1)
17:             Marcar la clique actual como la clique de mejor frontera hasta ahora   ▷ O(1)
18:
19:     return (CliqueDeMejorFrontera, MejorFrontera)
```

Considerando que el ciclo *while* del **Algoritmo 5** tiene $O(n)$ iteraciones en el peor caso (por ejemplo, en grafos completos donde todo nuevo nodo forma una clique con los anteriores), y que también en el caso de grafos completos $(\forall\ \text{i})\ \text{O}(\sum_{j \in vecinos(i)} d(j)) = O(m)$, la **complejidad final** de la heurística equivale a $\text{O}(n * m)$.

Veamos puntualmente por qué las complejidades son como especifica el pseudocódigo. Para ello, recordemos que el grafo está implementado con lista de adyacencias. Sea n la cantidad de nodos del grafo, m la cantidad de aristas, y d(i) el grado del nodo i:

- Calcular el nodo de grado máximo es O(n) pues recorremos los n nodos y nos quedamos con aquél que tenga más (o sea, grado más alto). Recordemos que el grado se obtiene en O(1) pues es la longitud de la lista que corresponde al nodo.
- La frontera de un nodo único es simplemente su grado, que se obtiene en O(1)
- Si tenemos guardado los nodos de una clique en un arreglo de booleanos (donde $a[i] = 1$ sii i pertenece a la clique) podremos saber en O(1) si un nodo está en la clique. Luego, saber si un nodo j es vecino de todos los nodos de la clique es equivalente a recorrer sus vecinos y ver si la cantidad de ellos que están en la clique es igual a la cantidad de nodos de la clique (dato que también puedo tener guardado fácilmente). Luego, este proceso se realiza en O(d(nodo)).
- Teniendo guardada la frontera exacta de una clique, podemos obtener fácilmente la frontera de ($clique \cup nodo_i$) pues es sumarle a la frontera el grado de $nodo_i$ (O(1)) y restarle dos veces la cantidad de nodos de *clique*. Se resta dos veces pues por un lado sabemos que $nodo_i$ es adyacente a todos los nodos de la clique, y por otro lado que habiendo agregado un nodo a la clique, la frontera anterior sumaba 1 por cada elemento de la clique adyacente a ese nodo.
- Observemos que quedarse con la mejor de todas las fronteras en cada iteración, así como obtener el último nodo agregado, son operaciones O(1) pues a nivel implementación en el medio del ciclo for voy actualizando a medida que encuentro una frontera mejor.

3.2 Máxima Clique Mayor Frontera

La heurística *MCMF* -máxima clique mayor frontera- se basa en suponer que una clique maximal, tiene más probabilidades de ser la clique de mayor cantidad de fronteras.
Elegimos implementar ésta heurística principalmente por tres motivos:

- Primero queríamos contrastar los resultados entre una heurística no convencional como ésta contra una heurística que se base en fundamentos mejores (con respecto al resultado a devolver), como *MFCGM*, con el sentido de confirmar que no cualquier heurística es buena.
- Segundo tenemos también que ésta heurística resuelve el peor caso de la heurística anterior, mostrando que por más que una heurística no sea convencional puede funcionar mejor que otras dependiendo del contexto, y resaltando lo incierto de los resultados que se pueden obtener a través de éstos métodos, y cómo influye en verdad el la entrada. También para marcar especialmente que las heurísticas no son algoritmos aproximados, por lo que no hay una lejanía fija del resultado óptimo.
- Por último consideramos que esta nueva heurística va a portar una muy buena semilla para luego aplicar el algoritmo de Búsqueda local. Queremos ver qué pasa cuando, por más que en un caso promedio devuelva peores resultados que la heurística de *MFCGM*, aplicándole la búsqueda cuánta mejoría puede haber en la solución final.

Al intentar buscar cliques maximales, vemos que encontrar la mayor de las cliques maximales no es un problema que se resuelva en tiempo polinomial. Entonces, debemos aplicar otra heurística golosa para resolver el problema.
Ésta se va a basar en ordenar el arreglo de nodos en orden descendente respecto al grado del mismo y, tomando cada nodo como inicial, ir iterando sobre el arreglo en el orden mencionado con el fin de formar la clique maximal que contenga a ese nodo. Nos vamos guardando la clique maximal con mayor frontera hasta ahora conseguida que devolveremos como respuesta. En cada iteración, vamos a ir comparando la clique guardada con la clique generada en ésta iteración, y en caso de que la última posea mayor frontera, se actualizará como nueva solución.

En resumen, la heurística se basa en:

- Ordenar el arreglo de nodos por grado
- Iterar sobre el arreglo de nodos del grafo
- Por cada iteración i, crear una clique maximal que contenga al nodo Vi
- Comparar las fronteras entre la clique de mayor frontera guardada anteriormente y la clique creada en la iteracion. Actualizar solución con la clique de mayor frontera.

Al finalizar estos pasos tengo una solución que se generó de forma golosa con un algoritmo de complejidad polinomial (se analizará más adelante).

3.2.1 Instancias no optimas

Las instancias de peor caso para esta heurìstica son los grafos completos.

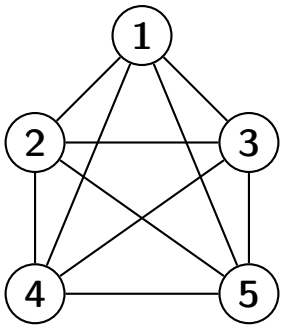


Figure 3: Instancia Completa K_5

En estas instancias, todas las cliques maximales calculadas serán exactamente *todo* el grafo completo. Al estar todas las aristas incluidas dentro de él, la frontera máxima obtenida será siempre 0. Dado esto, cualquier subgrafo incluido en él (incluyendo un nodo no aislado) conformará una clique de mayor frontera que la solución devuelta. Para peor, nuestra heurística de búsqueda de clique máxima **nunca va a fallar en estos casos**, y siempre devolverá como

solución la clique igual a todo el grafo.
Recordemos la forma de obtener la clique máxima implementada por nuestra heurística. Ahora, sin importar el orden en que estén ordenados los nodos (ya que todos tienen el mismo grado), partiendo de cualquier nodo raíz, se podrá agregar a la clique el nodo por el que se está iterando. Devolviendo como resultado la mayor clique todo el grafo.

3.2.2 PseudoCódigo y Complejidad teórica

Algorithm 4 MCMF

```
procedure MCMF(Graph grafo)
    res ← 0                                          ▷ O(1)
    vecRes ← 0                                       ▷ O(N)
    CMaux(grafo, res, vecRes)                        ▷ O(N^3)
    return (res, vecRes)
```

Algorithm 5 MCMFaux

```
procedure MCMFaux(Graph grafo, int res, vector<int> vecRes)
    NodosOrdenados = OrdenarPorGrado(V)              ▷ O(N^2)

    for i = 0 in NodosOrdenados do                   ▷ O(N)
        for j = 0 in NodosOrdenados do               ▷ O(N)
            if NodoFormaClique(j) then               ▷ O(|Vecinos(j)|)
                agregarAClique(j)
        if Frontera(Clique) > MejorFrontera then     ▷ O(N)
            MejorFrontera = Frontera(Clique);        ▷ O(1)
            Marcar clique actual como la clique de mejor frontera;  ▷ O(1)

    res = (CliqueDeMejorFrontera, MayorFrontera)     ▷ O(N^3)
```

En el **Algoritmo 10** se pueden divisar los pasos que describimos en la **sección 3.2**. Utilizando una representación con una lista de adyacencia, la cual nos permite obtener el grado del nodo en tiempo constante, la complejidad teórica final será $O(n^3)$

Veamos puntualmente por qué las complejidades son como especifica el pseudocódigo. Para ello, recordemos que el grafo está implementado con lista de adyacencias. Sea n la cantidad de nodos del grafo, m la cantidad de aristas, y d(i) el grado del nodo i:

- Primero ordenamos nuestro arreglo de nodos en tiempo $O(n^2)$. Cada comparación se hace en tiempo constante ya que podemos obtener d(i) en tiempo constante.
- La doble iteración cuesta $O(n^2)$
- Si tenemos guardado los nodos de una clique en un arreglo de booleanos (donde $a[i] = 1$ sii i pertenece a la clique) podremos saber en O(1) si un nodo está en la clique. Luego, saber si un nodo forma una clique, es decir saber si un nodo j es vecino de todos los nodos de la clique, es equivalente a recorrer sus vecinos y ver si la cantidad de ellos que están en la clique es igual a la cantidad de nodos de la clique (dato que también puedo tener guardado fácilmente). Luego, este proceso se realiza en O($|Vecinos(nodo)|$).
- Para saber si un nodo i forma clique es necesario verificar que todos los nodos que forman parte de la clique sean vecinos de nuestro nodo i. Sin embargo, la función esta implementada de tal manera que permite recorrer los vecinos del nodo una sola vez, recordando cuales de los vecinos son parte de la clique y comparando con la longitud de esta. Como la cantidad de vecinos esta acotado por n, el coste total es $O(n)$
- Finalmente se comparan la frontera de la clique maximal obtenida en la iteración y la guardada anteriormente, actualizando la solución.

3.3 Experimentación

En esta sección realizamos diversas experimentaciones:

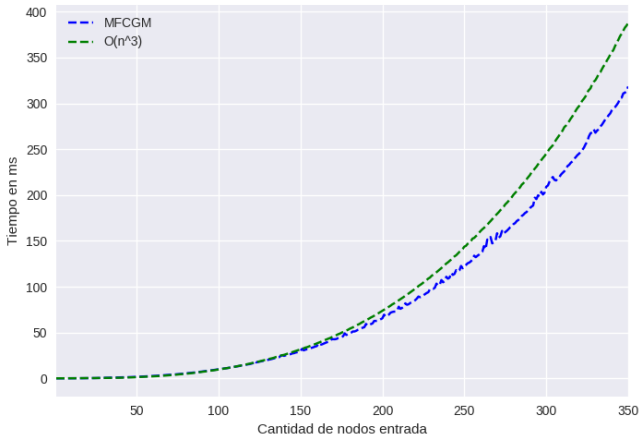
- Corremos ambas heurísticas con grafos completos (donde en particular *MFCGM* debiera tener un tiempo alto de cómputo) y comparamos sus tiempos de cómputo con las curvas de funciones $O(n^3)$ y $O(n^4)$ para verificar que se cumplen las complejidades.
- Comparamos la frontera obtenida y el tiempo de cómputo de cada heurística con el algoritmo exacto en grafos generados al azar según cada cantidad de nodos, fijando cierta densidad de aristas.
- Experimentación general para grafos al azar hasta n grande con cierta densidad fija para ver qué heurística devuelve una frontera mejor en general.

3.3.1 Tiempo de cómputo

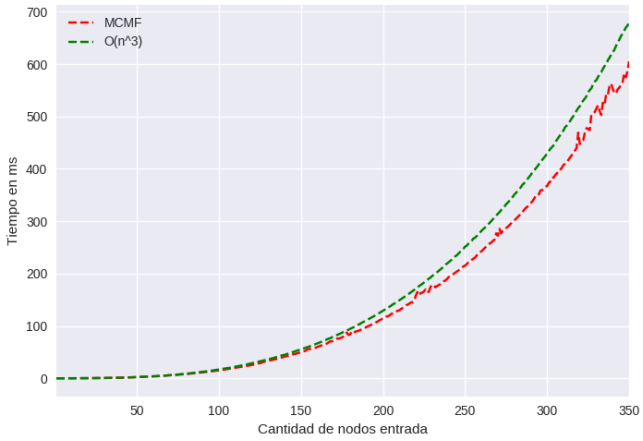
Fijando grafos completos como entrada, sea n la cantidad de nodos. Para cada n desde 1 hasta 350 se corrieron 40 repeticiones de cada heurística y se tomó el promedio de los tiempos de cómputo según cada n.

A su vez, se creó una función que represente correctamente el tiempo de computo de la complejidad $O(n^3)$ -3 ciclos anidados que se recorran n veces- y para éstas también se tomó el promedio de 30 repeticiones para cada n.

Se eligieron grafos completos para esta experimentación pues esa familia de grafos maximiza la complejidad teórica de cada una de las heurísticas.



(a) Figure 4: MFCGM vs $O(n^3)$

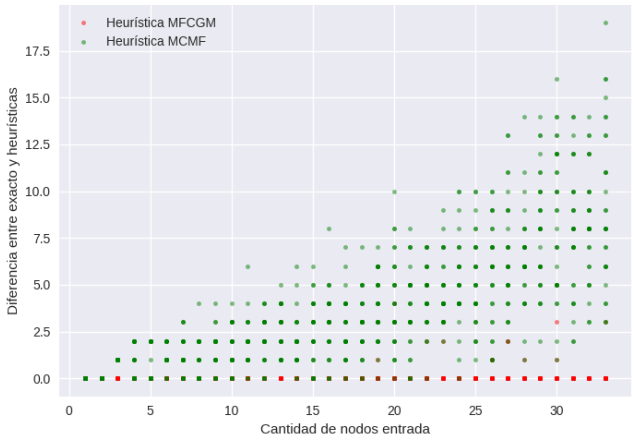


(b) Figure 5: CM vs $O(n^3)$

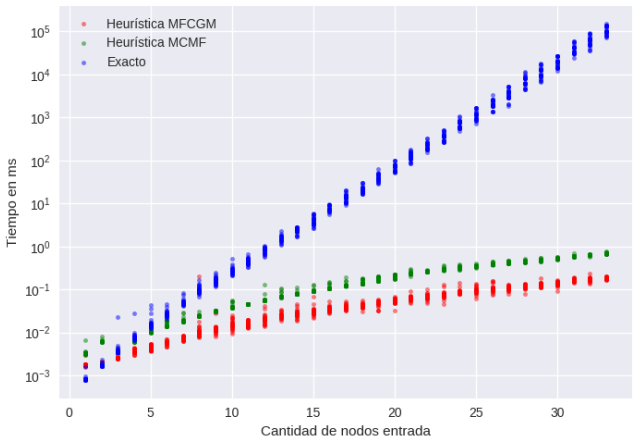
Analizando las **Figuras 4 y 5** podemos ver como en cada caso las curvas generadas por la ejecución de las heurísticas son similares a las curvas de las funciones que representan su complejidad (en cada caso con una constante distinta para la función $O(n^3)$, siendo 4 y 7 las mismas para *MFCGM* y *MCMF* respectivamente).

3.3.2 Heurística 1 vs Heurística 2 vs Algoritmo Exacto

Para estos tests generamos instancias al azar. Sea n la cantidad de nodos, para n entre 1 y 33 creamos 30 instancias de forma aleatoria con el generador de grafos descripto en el final del informe. Luego, en ésta experimentación todo grafo tendrá una densidad de aristas de $0,5$. Para cada n se graficó por un lado la diferencia entre el resultado de cada heurística y el resultado del algoritmo exacto en cada una de sus instancias (30 puntos distintos para cada n) y por otro lado el tiempo de cada una de esas instancias (para cada uno de los algoritmos). La finalidad del test es exponer que efectivamente las heurísticas no devuelven siempre la solución exacta, y también que quede claro que obtener la solución exacta es muchísimo más costoso temporalmente.



(a) Figure 8: Diferencia entre Heurísticas y Exacto Grafos aleatorios Res



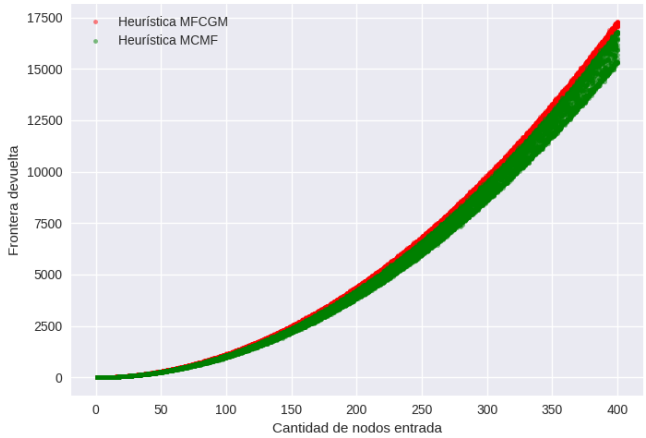
(b) Figure 9: Heurísticas vs Exacto Grafos aleatorios tiempo

Vemos en la **Figura 9** que los tiempos de cómputo no nos dieron ninguna sorpresa. El algoritmo exacto tiene un crecimiento de carácter exponencial mientras que las heurísticas no. Por otro lado, pese a ser ambas de la misma complejidad teórica, la heurística *MFCGM* tiene menor tiempo de cómputo en todos los casos. Si bien éstos casos son pequeños y no sirven para un análisis temporal profundo, esto se contrasta con el experimento **3.3.1**

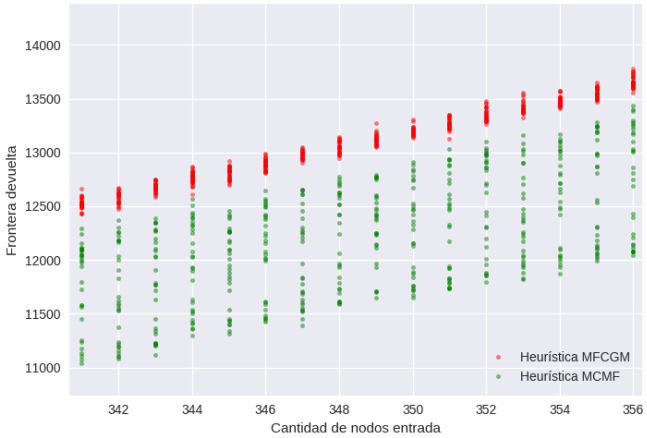
Analizando la **Figura 8** puede verse que aun para cantidades de nodos pequeñas en general la heurística *MCMF* devuelve soluciones peores que el algoritmo exacto. Con respecto a la heurística *MFCGM*, para instancias pequeñas pareciera que, salvo en el caso de encontrarnos con grafos muy particulares (como el descripto en **3.1.1**), ésta heurística devuelve en casi todas las instancias la solución correcta.
Ésta misma figura nos permite plantear la hipótesis de que el resultado obtenido por la heurística *MCMF* en general es peor que el que da la heurística *MFCGM*. Sobre ésta hipótesis se tratará el siguiente experimento.

3.3.3 Experimentación general

Aquí también utilizamos, como en el test anterior, el generador de grafos. Fijamos una densidad de $0,5$ y experimentamos para n entre 1 y 400 ambas heurísticas con 30 instancias aleatorias para cada n y graficando para cada una su resultado y tiempo de cada heurística. El objetivo de ésta experimentación es intentar definir qué heurística nos resulta mejor en general.



(a) Figure 11: Resultado heurísticas Grafos aleatorios



(b) Figure 12: Zoom Figure 11 cantidad de nodos 340 a 358

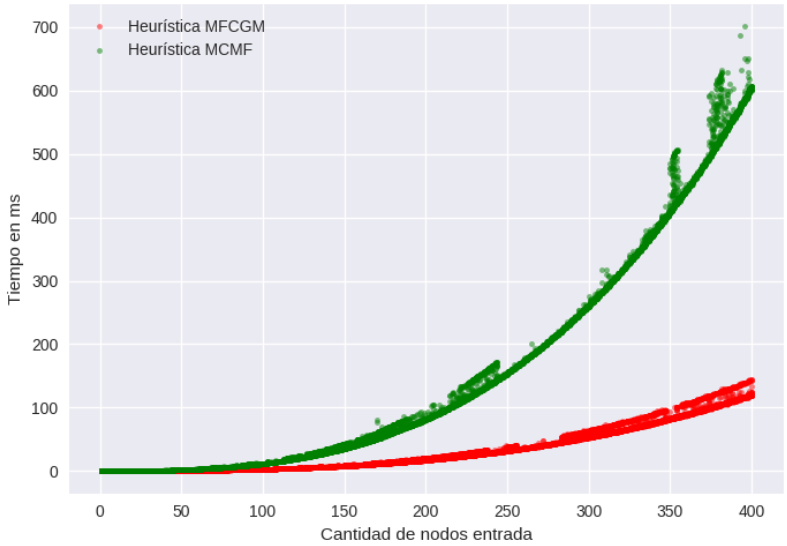


Figure 7: Figure 13: Tiempo Heurísticas Grafos aleatorios

Aquí los resultados fueron como supusimos tanto al plantear la heurística *MCMF* (pues en realidad es una heurística para resolver otro problema), como al realizar el experimento **3.3.2**. En general con respecto a la frontera devuelta *MFCGM* devuelve soluciones mejores que *MCMF*, lo que queda claro analizando la **Figura 11**. Por si no se entiende tanto el gráfico, mostramos en la **Figura 12** una parte del mismo con zoom (entre n 340 y 358 exactamente), en la que se ve claramente que la heurística *MFCGM* devuelve soluciones mejores en casi todos los casos.

Con respecto al tiempo de cómputo, vemos en la **Figura 13** que la heurística *MFCGM* corre más rápido que *MCMF*, lo que tiene sentido pues la complejidad teórica de ésta última es peor.

Por lo expuesto, concluimos que en general la heurística *MFCGM* es la mejor de las dos.

4 Heurística Búsqueda Local

En esta sección se explica la heurística de búsqueda local utilizada. La misma se basa en:

Dada una solución posible al problema, definir un conjunto de soluciones vecinas de magnitud polinomial. Luego buscar cuál solución es mejor entre todas las vecinas y volver a aplicarle el algoritmo a ésta (hasta que no haya una mejor). En otras palabras básicamente vamos yendo entre soluciones vecinas hasta encontrar alguna que sea la mejor localmente, por lo que no me convendrá seguir por algún vecino.

Es importante que el conjunto definido para una solución sea de magnitud polinomial ya que justamente en una heurística tratamos de resolver problemas cuya solución exacta es de carácter exponencial. Si nosotros no acotamos a éstos términos el conjunto de vecinos, no habría punto en ejecutar esta heurística.

Lo positivo de la búsqueda local es que se puede combinar con otras heurísticas ya planteadas para mejorar sus soluciones, por ejemplo dada una heurística golosa que te devuelve una solución posible, se empieza el algoritmo de búsqueda local a partir de este resultado y finalmente vamos a terminar encontrando un resultado mejor o igual al recibido.

Como lado negativo, obviamente como toda heurística no devuelve un resultado exacto. El algoritmo como describimos se basa en encontrar algún pozo dentro de las soluciones, pero no en encontrar el mejor de todos. Si en alguna iteración sucede que encontramos un máximo local -pero no máximo global- el algoritmo va a parar su búsqueda y retornar un resultado no óptimo.

En lo que respecta a nuestro algoritmo de búsqueda local definimos dos diferentes heurísticas, las cuáles difieren en la cardinalidad del conjunto de vecinos para cada solución. La que vamos a llamar Vecindad Lineal define un conjunto de vecinos de orden $O(n)$ para cada solución posible, mientras que la que vamos a llamar Vecindad Cuadrática toma un conjunto de vecinos de orden $O(n^2)$. Vamos a explicar cada algoritmo y sus diferencias en las siguientes secciones.

4.1 Vecindad Lineal

En nuestra primer heurística de búsqueda local definimos, como dijimos previamente, una vecindad O(n) (con n la cantidad de nodos del grafo).

Dado un conjunto soulcion del problema, definimos a sus vecinos como el conjunto

N(S) igual a
$\{S - \{v\} \ : \ v \in S \ \wedge \ EsClique(S - \{v\})\}$

$\cup$
$\{S + \{v\} \ : \ v \notin S \ \wedge \ EsClique(S + \{v\})\}$

Luego el algoritmo se basa, en simplemente tomar al elemento de máximas fronteras en $N(S)$, y en caso de superar las fronteras de S, se vuelve a aplicar la busqueda local a partir de ahí. En caso contrario el algoritmo se frena y retorna las fronteras de S como resultado obtenido.

4.2 Vecindad Cuadrática

Tanto la idea como la implementación de esta heurística son muy similares a la previamente vista. En lo que se diferencian simplemente es que el conjunto de vecinos por cada solución S del problema aumenta.

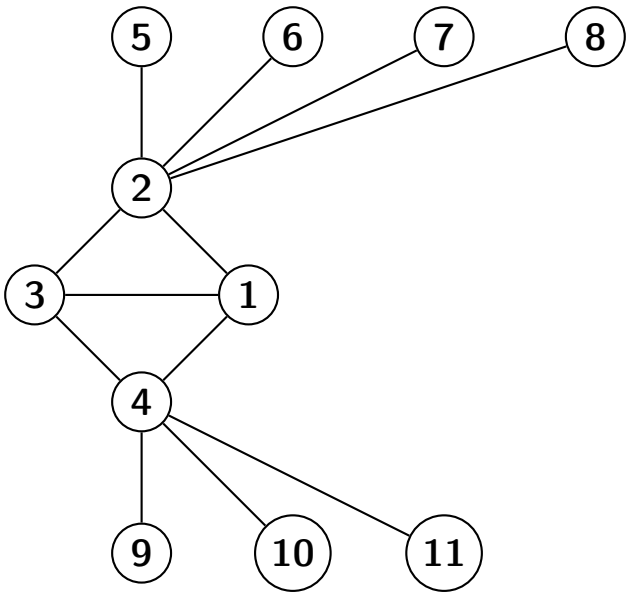
El nuevo conjunto $N(S)$ se define como :
$\{S - \{v\} \ : \ v \in S \ \wedge \ EsClique(S - \{v\})\}$

$\cup$
$\{S + \{v\} \ : \ v \notin S \ \wedge \ EsClique(S + \{v\})\}$
$\cup$
$\{S - \{v\} + \{u\} \ : \ v \in S \ \wedge \ u \notin S \ \wedge \ EsClique(S - \{v\} + \{u\})\}$

Éste conjunto de vecinos tiene cardinalidad de orden cuadrático con respecto a los nodos del grafo, porque en el peor caso, cuando G es un grafo completo, pasa que para cada nodo v dentro de S, v puede ser swapeado por cualquiera de los nodos que no se encuentran ya en S, logrando sumar n vecinos por cada vértice dentro de S y una complejidad espacial de orden $O(n^2)$

En cuanto a su implementación algorítmica no suma ninguna complejidad agregar esta condición, puesto que sólo hay que considerar un nuevo conjunto de soluciones posibles a las cuáles moverse, pero la idea general permanece intacta.

Esta idea surgió para evitar casos como el siguiente:



Aquí si la solución semilla es el K_3 formado por los nodos $[1, 3, 4]$, al ejecutar la Búsqueda Local definida con vecindad Lineal hubiese devuelto la clique formada por los nodos $[1, 4]$ con 6 aristas fronterizas. Mientras que la búsqueda Local definida con Vecindad Cuadrática hubiese considerado el caso de eliminar el nodo 4 de la solución y agregar al nodo 2, el cual mejoraba el escenario. Y luego de ahí restar el nodo tres, dando como resultado final a la clique $[1, 2]$ con un total de 7 aristas fronterizas.

4.3 Implementación

Utilizamos como representación del conjunto $SolAct$, un arreglo del tamaño n, en el cual la posicion i es igual a 1 si y solo si el nodo i pertenece a la $SolAct$, e igual a 0 en caso contrario.

4.3.1 Vecindad Lineal

Para implementar el algoritmo elegimos una estructura levemente diferente a las vistas en la materia. Utilizamos una representación de grafos de Arreglos de árboles logarítmicos (estructura Set de c++).

El por qué de esta elección está explicado más adelante pero para ello primero tenemos que mostrar el pseudo código de nuestra heurística:

```
1: procedure BusquedaLocalLineal(In G: Arreglo<Conj<int», In/Out SolAct: Conj<int>)→int
2:     bool NoEncontréPozo = true
3:     while NoEncontréPozo do
4:         for x ∈ [1..n] do                                              ▷ O(n * (n + m))
5:             if x ∉ SolAct ∧ esVecinoDeTodos(SolAct, x) then            ▷ O(n + m))
6:                 NodosQuePuedoSumar.Add(x)
7:         for x ∈ [1..n] do                                              ▷ O(n)
8:             if x ∉ SolAct then                                         ▷ O(1)
9:                 NodosQuePuedoRestar.Add(x)
10:        for x ∈ NodosQuePuedoSumar do                                  ▷ O(n^2)
11:            if SiMejoraLaSolucionesDeSumar(SolAct + x) then            ▷ O(n)
12:                NodoQueMejoraAlSumar = x
13:        for x ∈ NodosQuePuedoRestar do                                 ▷ O(n^2)
14:            if SiMejoraLaSolucionesDeRestar(SolAct + x) then           ▷ O(n)
15:                NodoQueMejoraAlRestar = x
16:                                  ▷ Ahora queda simplemente ver Cual es la mejor
17:        if SiNingunaFronteraSuperaALaActual then                       ▷ O(1)
18:            NoEncontrePozo = false                                     ▷ O(1)
19:        else if SiLaMejorFronteraEsSumandoUnNodo then                  ▷ O(1)
20:            SolAct.Add(NodoQueMejoraAlSumar)                           ▷ O(log(n)
21:        else
22:            SolAct.Erase(NodoQUeMejoraAlRestar)                        ▷ O(log(n))
23:    endWhile
24:    return CalcularFronteras(G,SolAct)
```

Explicacion:

1. Iteramos todos los nodos, y por cada uno que no pertenezca a $SolAct$, iteramos los nodos del conjunto $SolAct$. Recorriendo sus vecinos en $O(d(n))$. Chequeamos que en cada conjunto de vecinos aparezca el nodo que quiero sumar. Esto como la suma de los grados esta acotada por m, lo logramos en $O(m)$. Y por ultimo de ser vecino de todos los nodos de $SolAct$, lo agregamos al conjunto $NodosQueSePuedenSumar$
2. Copiamos el conjunto $SolAct$ en $NodosQueSePuedenRestar$
3. Para cada nodo en el conjunto $NodosQueSePuedenSumar$, se lo agrego a la solución y calculamos su fronteras en $O(n)$, ya que por cada nodo sabemos que la cantidad de fronteras que suma a la solución es de $d(n_x - |SolAct|)$. En caso de ser la mejor hasta el momento lo salvo en $NodoQueMejoraAlSumar$.
4. Analogo al anterior paso, salvo que recorremos $NodosQueSePuedenRestar$
5. Por último chequeo si la solución mejora llendo por algún vecino. En caso de ser asi volvemo a repetir el algoritmo con la nueva semilla. Y en caso contrario se frenan las iteraciones y se devuelve $SolAct$

Entonces como podemos ver dentro de cada iteración ejecutamos varias operaciones de complejidad temporal alta, por lo que buscamos una estructura que logre optimizar la complejidad total. Para ello hicimos el siguiente análisis.

Las complejidades de cada una de estas operaciones con otras estructuras serían:

	MatrizDeAdyacencia	ArregloDeListas	ArregloDeEstructSet
Item 1	$O(n^3)$	$O(n(n+m))$	$O(n*(n+\sum log(d_i)))$
Item 2	$O(n)$	$O(n)$	$O(n)$
Item 3	$O(n^3)$	$O(n^2)$	$O(n^2)$
Item 4	$O(n^3)$	$O(n^2)$	$O(n^2)$

Al fin de cuentas podemos ver que la representación que nos reducía mayormente la complejidad asintótica del algoritmo es la representación de un Arreglo De Sets

La complejidad en peor caso de los algoritmos de Búsqueda Local no se puede acotar ya que éstos siguen ejecutando hasta que encuentre un pozo dentro del campo de soluciones que recorren. Sin embargo bien se puede estimar la complejidad temporal dentro de cada iteración del algoritmo, por lo que resultamos en lo siguiente:

COMPLEJIDAD POR ITERACIÓN: Como mostramos dentro del pseudocódigo la operación ejecutada de mayor complejidad dentro de cada iteración es de $O(n^2 + n * m)$, debido al ciclo que se ejecuta para buscar los nodos que se le pueden agregar a la solución semilla.

4.3.2 Vecindad Cuadrática

Su implementación es muy similar a la ya provista para la Vecindad Lineal. El único agregado es que se crea una nueva variable de mejor vecino a tener en cuenta, en representación a los vecinos que se forman al swapear dos nodos de G.

Esta se calcula de la siguiente forma:

```
for i ∈ NodosFueraDeSolAct do                                   ▷ O(n² * (n + m))
    for j ∈ SolAct do                                           ▷ O(n * (n + m))
        if EsVecinoDeTodos(SolAct − j, i) then                  ▷ O(n + m)
            NodosQUePUedoSumarYRestar.Add(< i, j >)

for x ∈ NodosQuePuedoSumarYRestar do                            ▷ O(n³)
    if SiMejoraLaSolucionesDeSwapear(SolAct - x.first + x.second) then   ▷ O(n)
        NodosQueMejoranSwapear = <x.first, x.second>
```

Luego podemos ver que el aumento de vecindad trae consigo un aumento en cuanto a la complejidad temporal que conlleva cada iteración de la búsqueda, dando finalmente una complejidad total de $O(n^2 * (n + m))$.

4.4 Experimentación

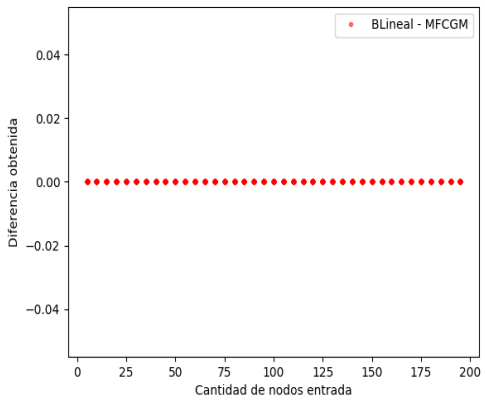
Los experimentos de esta sección se hicieron utilizando el mismo generador de grafos, que para la sección **3.3**. Cada experimento fue ejecutado evaluando los resultados con ambas vecindades con el fin de encontrar si el poder de computo que se cede al plantear una vecindad más grande logra una mejora significativa en el resultado.

En la experimentación buscamos encontrar distintas características que distingan a cada vecindad. Para ello nos fijamos su comportamiento evaluando con densidades de aristas fijas -la cual fuimos moviendo entre experimentos-. Buscamos ver cuántas iteraciones promedio ejecutaba cada algoritmo, la curva de complejidad temporal promedio que demostraba, y por último qué tan óptimo era su resultado.

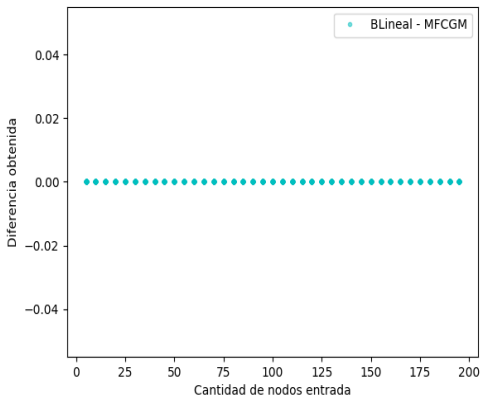
Búsqueda Local Implementada con Semilla de algoritmo MFCGM Sobre Grafos Random

En este experimento planeamos comparar los resultados de la heurística golosa MFCGM, aplicada sobre grafos generados de forma random, contra el resultado de aplicar a la respuesta de este mismo algoritmo la búsqueda local.

Para ejecutar esto hicimos una experimentación donde probamos para cada n entre 5 y 100, 30 instancias distintas, graficando la diferencia entre los resultados obtenidos por cada algoritmo para cada instancia.



Diferencias Entre Heuristica MFCGM y Busqueda Cuadratica Con Dicha Semilla



Diferencias Entre Heuristica MFCGM y Busqueda Lineal Con Dicha Semilla

Como podemos ver en este gráfico notamos por primera vez que los algoritmos de Búsqueda Local Lineal, Cuadrática y la Heuristica MFCGM devuelven exáctamente los mismo resultados. ¿Por qué pasa esto? Lo veremos de a partes.

1. Primero vamos a analizar la heurística MFCGM intentando descubrir por qué se da este fenómeno

El algoritmo de MFCGM, en lineas generales (como fue descripto en su sección) va agregando a la clique resultado los nodos vecinos del nodo de mayor grado, en orden del de mayor grado hacia el de menor grado, de forma que mantengan una clique entre sí y vayan mejorando la solución.

Queremos aclarar por qué afirmamos que los va agregando en orden de mayor grado a menor grado. El algoritmo verdaderamente se fija en todos los vecinos del nodo de mayor grado sin distinciones entre sí, y calcula con cada uno cuál es la frontera obtenida -si el conjunto de dos nodos era una clique-. Luego elige al que obtuvo la mayor cantidad de aristas fronterizas entre todos.

Sin embargo estas dos operaciones son equivalentes: Asumamos que tenemos dos nodos n_1 y n_2 tales que el grado de n_1 es mayor al de n_2, ambos son vecinos del nodo de máximo grado (n_m), no pertenecen a la solución actual, y la solución actual agregando n_1 o n_2 sigue siendo una clique. Entonces queremos ver que el conjunto $SolACt + n_1$ va a tener mayor cantidad de aristas fronterizas que $SolAct + n_2$.

Tenemos que la cantidad de aristas fronterizas que va a aportar cada nodo a la solución es de $g(n_x) - 2|SolAct|$, lo cual se deduce trivialmente porque cada arista que se conecte con un nodo dentro de $SolAct$ no va a ser fronteriza. Por lo que se reduce en la simple cuenta, que si se cumple la desigualdad $g(n1) > g(n2)$, tenemos que $g(n1) - 2|SolAct| > g(n2) - 2|SolAct|$, lo cuál determina que el algoritmo de MFCGM elija primero al nodo vecino de n_m de mayor grado.

2. Ahora, con las nuevas conclusiones sobre MFCGM queremos responder por qué la heurística es un máximo local con respecto a la vecindad marcada por nuestra Vecindad Lineal.

Como dijimos anteriormente la Vecindad Lineal marca como vecinos a todas las soluciones que contengan un nodo más o un nodo menos que la solución actual. Entonces, hay que ver que MFCGM es mayor que todas las soluciones vecinas:

- *Si la solución vecina contiene un nodo menos.* Como mencionamos recientemente, la heurística golosa de la cual estamos hablando sólo agrega un nodo a su solución si cumple particularmente que la cantidad de aristas fronterizas aumentan a las del conjunto $SolAct$. Por lo que es directo decir que si le restas el último nodo agregado al conjunto solución, que devuelve el algoritmo MFCGM. Por invariante de cómo fue armado el conjunto, esto empeora la solución.
  Ahora como dijimos en la observación anterior, el último nodo agregado es al mismo tiempo el de menor grado entre los vecinos de n_m. Y tenemos que la cantidad de fronteras que agrega un nodo a la solución, es de $d(n_x) - 2|SolAct|$.
  Luego la cantidad de fronteras de la solución tras haberle restado un nodo es igual a $CantFronteras(SolAct) - (d(n_x) - 2|SolAct|) + |SolAct| - 1$. Y de esto se deduce directamente, que si el grado de n_x aumenta, la cantidad de fronteras de la solución obtenida al restarle n_x es menor. Por lo que si este coeficiente ya no favorecía para restar a n_x, igual al nodo vecino de de n_m de menor grado, menos lo va a ser para cualquiera del resto, que poseen un grado mayor.
- *Si la solución vecina contiene un nodo más.* Para ver este caso acudimos a cómo actúa la heurística golosa. Esta, justamente en cada iteración del algoritmo chequea a todos los nodos que no pertenecen a la solución actual, y pueden formar una clique con la misma. Y luego elige al mejor entre los nodos que agregandolos a la solución aumentan la cantidad de fronteras agregandolos. En caso de no habe ninguno el algoritmo termina. En consecuencia, si hubiese un nodo tal que efectivamente sucede que sumandolo a la solución actual, mejora la cantidad de fronteras, genera un absurdo a como funciona la heuristica golosa. Porque en la última iteración no hubiese frenado y habría agregado un nodo más al menos.

3. Finalmente, debemos ver por qué los resultados del algoritmo de Búsqueda Local definido con Vecindad Cuadrática no mejoran la solución provista por la heurística golosa MFCGM.

Como dijimos en secciones anteriores, la búsqueda local definida con Vecindad Cuadrática simplemente agrega a los vecinos de una solución, a aquellas soluciones del problema que se formen de restar y agregar un nodo. Luego, nos queda justificar por qué con esta nueva vecindad el resultado de MFCGM sigue permaneciendo como máximo local. Para probarlo dividiremos en dos casos:

- *Si sacamos al nodo máximo n_m.* Queremos ver si suponemos que tendremos una nueva solución con frontera más grande al sacar a n_m y reemplazarlo por otro nodo diferente que sea compatible con el resto de los nodos de $SolAct$, vamos a llegar a un absurdo. Esto sale directamente con el coeficiente que planteamos anteriormente de la cantidad de fronteras que un nodo puede sumar a la $SolAct$, que era igual a $g(n) - |SolAct|$, y como tenemos que n_m es el nodo de mayor grado dentro del grafo, es evidente que no hay ningún otro nodo que eleve este coeficiente a uno más elevado que él. Por lo que el escenario de querer sacarlo no se podría de dar nunca.

- *Si sacamos cualquier otro nodo.* Aquí la explicación de por qué no hay escenario donde nos convenga sacar un nodo y agregar a otro se deriva de lo explicado en el primer punto de esta sección donde aclaramos que los nodos que se suman a la solución del algoritmo MFCGM, se suman en orden de sus grados. Por lo que todo nodo que no pertenezca a la solución y sea vecino de n_m va a tener menor o igual grado al nodo de menor grado dentro de la solución, y nuevamente llegamos a un absurdo si planteamos que se aumentan la cantidad de fronteras al swapear un nodo de mayor grado por uno de menor grado dentro de la solución.

Finalmente como vemos, los resultados de aplicar búsqueda local con vecindad Lineal Y Cuadrática a la semilla provista por el algoritmo MFCGM son los esperados.

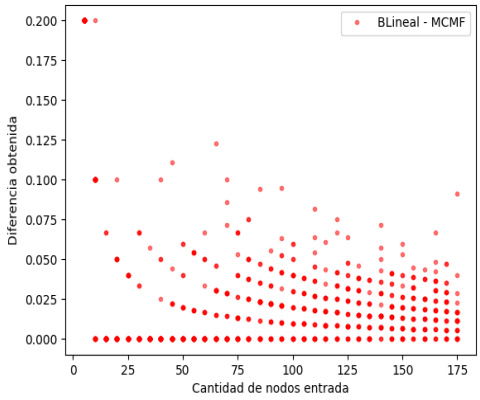
Búsqueda Local Implementada Con Semilla Heurística MCMF Sobre Grafos Random

En este experimento planeamos comparar los resultados de la heurística golosa MCMF aplicada sobre grafos generados de forma random contra el resultado de aplicar a la respuesta de este mismo algoritmo la búsqueda local. Para esto graficamos las diferencias relativas entre los resultados obtenidos por cada algoritmo, viendo así si con el aumento de aristas y nodos se producirá un cambio en la mejora del resultado proporcional al tamaño de la entrada.

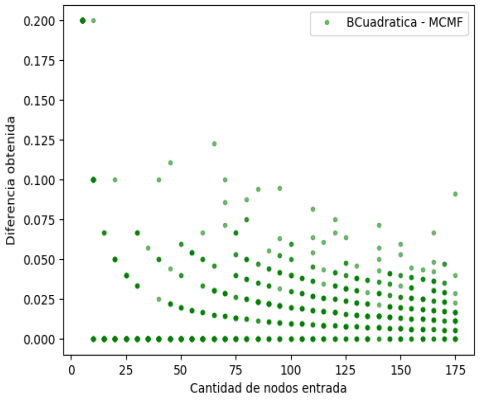
El experimento lo reprodujimos utilizando 3 densidades de manera fija. Con densidad baja (igual a 0.1), con densidad media igual a 0.5, y con densidad alta (igual a 0.8), con el fin de encontrar características particulares en el comportamiento de estos experimentos.

Para ejecutar esto hicimos una experimentación donde probamos para cada n múltiplo de 5 entre 5 y 200, 30 instancias distintas.

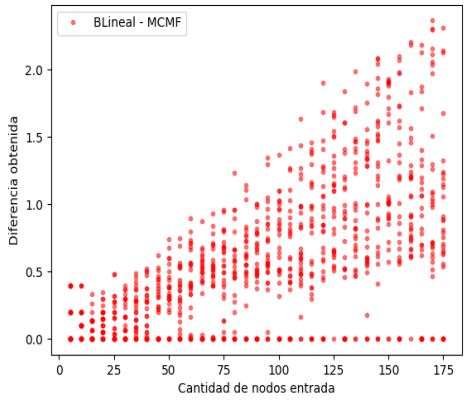
Diferencias Relativas entre los resultados de Búsqueda Lineal y Búsqueda Cuadrática vs Algoritmo MCMF



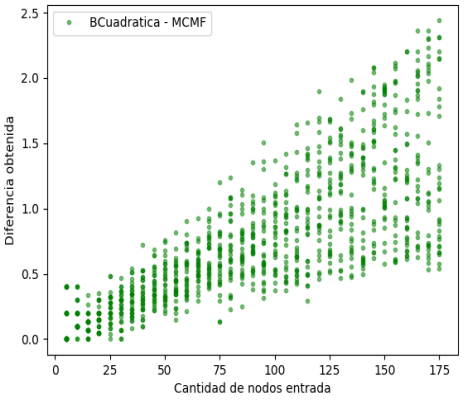
Grafos random con Densidad 0.1



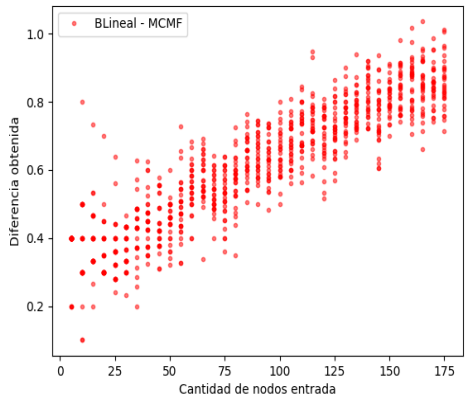
Grafos random con Densidad 0.1



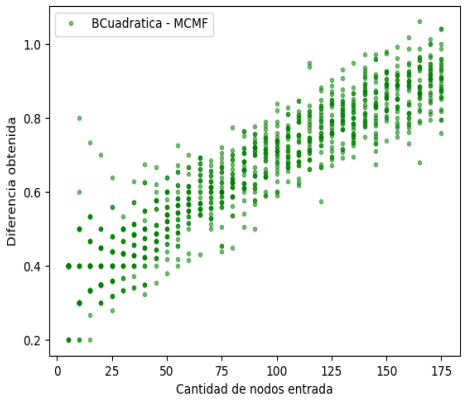
Grafos random con Densidad 0.5



Grafos random con Densidad 0.5



Grafos random con Densidad 0.8



Grafos random con Densidad 0.8

Como se puede observar en este experimento en primer lugar no aparecieron sorpresas mayores. Como esperábamos, la heurística de buscar cliques maximales con el fin de obtener la mayor cantidad de fronteras posibles se aleja significativamente de ser un máximo local en situaciones generales, y se ve cómo efectivamente se puede explotar el algoritmo de Búsqueda Local en estos términos.

Sin embargo podemos encontrar comportamientos diferentes del resultado del algoritmo a medida que se fueron cambiando las densidades de los grafos. En un principio, como muestra el primer gráfico, cuando la densidad de los grafos es igual a 0.1, a medida que se fueron aumentando el tamaño de los grafos la mejoría que aporta la búsqueda local fue decayendo. Suele ser por demás un valor muy cercano al devuelto por la heurística golosa *MFCGM*.

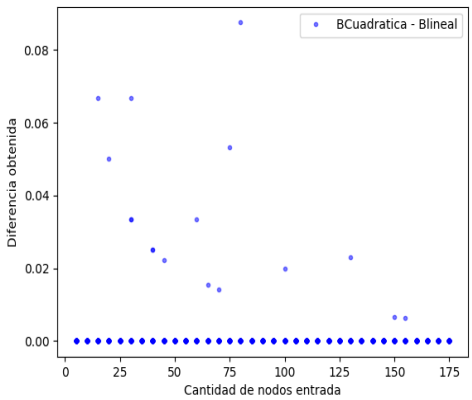
Por otro lado, analizando el gráfico representativo de la densidad 0.5, vemos que al aumentar la densidad de las aristas la búsqueda local empieza a mejorar la respuesta en relación a la cantidad de nodos, casi linealmente en los mejores casos. Aunque por contra parte mostrando una mayor dispersión entre las diferentes instancias.

Por último cuando la densidad es de 0.8 nuevamente se muestra que con el crecimiento de los nodos hay una mejora proporcional de las respuestas obtenidas, en particular con la propiedad de ser más conciso que en densidad 0.5.

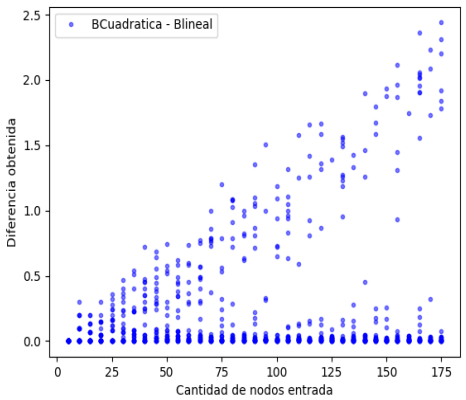
Finalmente cabe rescatar que la búsqueda local en grafos de densidad 0.8 por más que muestre una mejora proporcional al crecimiento de los nodos, se ve levemente más acotada que en los mejores casos de densidad 0.5.

Otro elemento para analizar es la similitud de los comportamientos. Anteriormente hicimos un análisis general ya que en ambos la evolución de las características se modifica de igual manera. Pero justamente queremos ver si el aumento de vecinos además trajo una mejoría por su cuenta en los resultados. Para ello planteamos los siguientes gráficos, mostrando las diferencias entre la heurística de búsqueda local lineal y cuadrática.

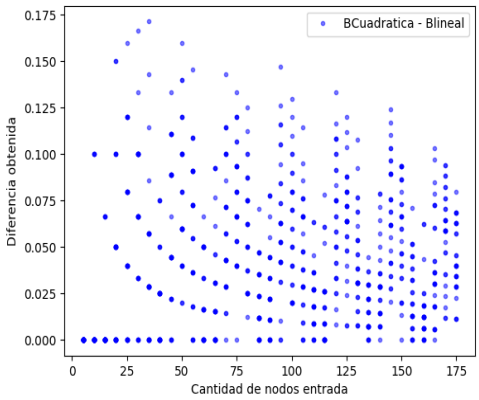
Diferencias Relativas entre los resultados de Búsqueda Lineal vs Cuadrática



Grafos random con Densidad 0.1



Grafos random con Densidad 0.5



Grafos random con Densidad 0.8

Como podemos ver, en primer lugar con densidad baja la diferencia no es significativa, lo cual es esperado puesto que en los casos en que la búsqueda cuadrática mejora son muy particulares y se dan en menor medida cuando la densidad es baja.

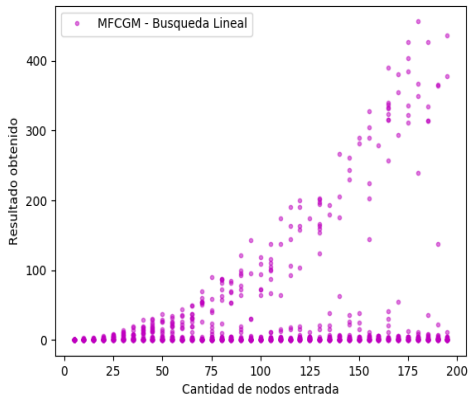
Siguiendo al siguiente gráfico, volvemos a notar la dispersión entre las mejorías a medida que el tamaño del grafo aumenta. Sin embargo dentro de los mejores resultados, se observa una fuerte mejora del resultado en relación al tamaño de la entrada.

Por último, con los grafos de densidad más alta, los resultados devueltos por ambas heurísticas se muestran menos distantes entre sí. Esto da una idea de cuándo es conveniente utilizar una heurística u otra, ya que como veremos a continuación el tiempo de cómputo entre ambos algoritmos difiere considerablemente (con grafos de densidad alta).

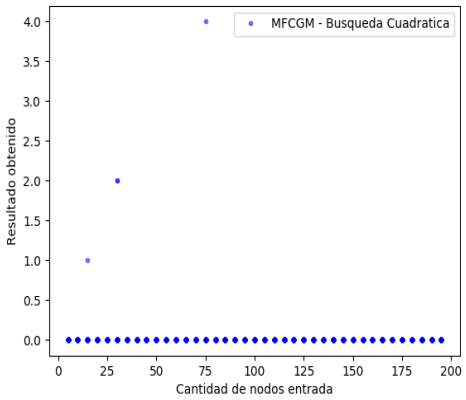
Experimento Comparación Contra Heuristica MFCGM

Como vimos en las últimas instancias, las heurísticas de búsqueda local lograron mejorar el resultado obtenido por el algoritmo MCMF significativamente. Ahora queremos ver cuán grande fue la mejoría comparándolo con la primer heurística golosa MFCGM.

Para ello utilizamos los mismo grafos creados para toda esta sección, de densidad 0.5. Graficaremos la diferencia entre los resultados de ambos para distinguir bien los puntos en los cuales devolvían un resultado muy parecido.



Diferencias Heurística MFCGM
vs
Búsqueda Lineal con semilla de MCMF



Diferencias Heurística MFCGM
vs
Búsqueda Cuadrática con semilla de MCMF

De estos resultados terminamos definiendo que la heurística MCMF no es una buena heurística. Una de las razones por las cuales nos interesó crearla y probar sobre ella era justamente que iba a aportar un buen escenario para utilizar la búsqueda local. Como demostramos a través de la experimentación, efectivamente así fue. Sin embargo, como vemos en los resultados del último test, la mejora obtenida en los mejores casos no llega a superar nunca a la primer heurística golosa, e incluso en los peores casos devuelve un resultado mucho peor. Por esto y también por el tiempo de cómputo que diferencia a estas heurísticas, concluimos que no es una buena decisión su utilización.

4.4.1 Experimento 2 - Comportamiento Temporal

Recordamos que anteriormente en la sección de implementación mencionamos que las complejidades por iteración de cada búsqueda local eran de:

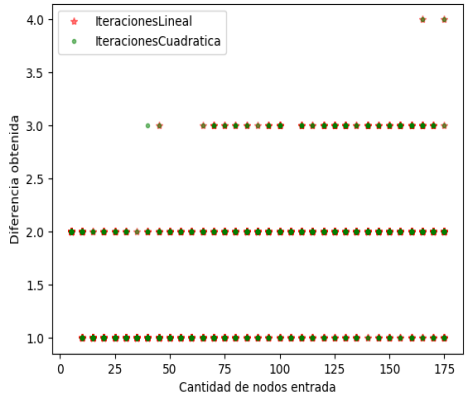
- Búsqueda Lineal : $O(n * (n + m))$
- Búsqueda Cuadrática: $O(n^2 * (n + m))$

Por lo que podemos notar que el comportamiento temporal es dependiente de dos variables: la cantidad de aristas y de nodos. Para ellos nuevamente ejecutamos los experimentos fijando densidades, y comparando los comportamientos entre los resultados.

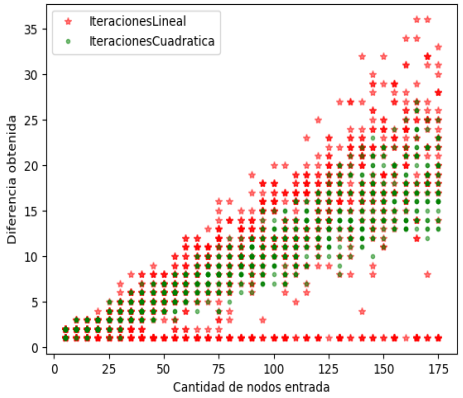
Como antes, para los experimentos de esta sección probamos para todo n múltiplo de 5 entre 5 y 200, 30 instancias, y graficamos todos los resultados obtenidos:

Experimento Cantidad de iteraciones

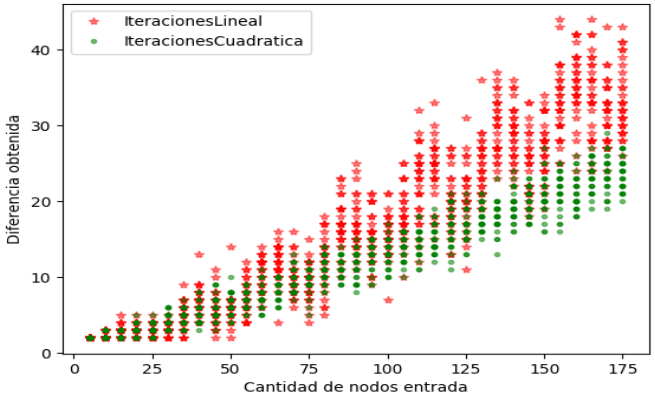
Para el primer experimento medimos la cantidad de iteraciones que ejecuta cada algoritmo para cada instancia.



Iteraciones en Grafos random con Densidad 0.1



Iteraciones en Grafos random con Densidad 0.5



Iteraciones en Grafos random con Densidad 0.8

De estos gráficos pudimos sacar una conclusión no esperada.Como vemos es que la BL Lineal suele emplear más iteraciones para alcanzar un pozo, que la BL Cuadrática.

Con densidades bajas como el cardinal del conjuntos de vecinos se acota bastante, esta diferencia no se muestra. Sin embargo esta característica se puede ya notar sin problemas en los grafos de densidad 0.5 y 0.8.

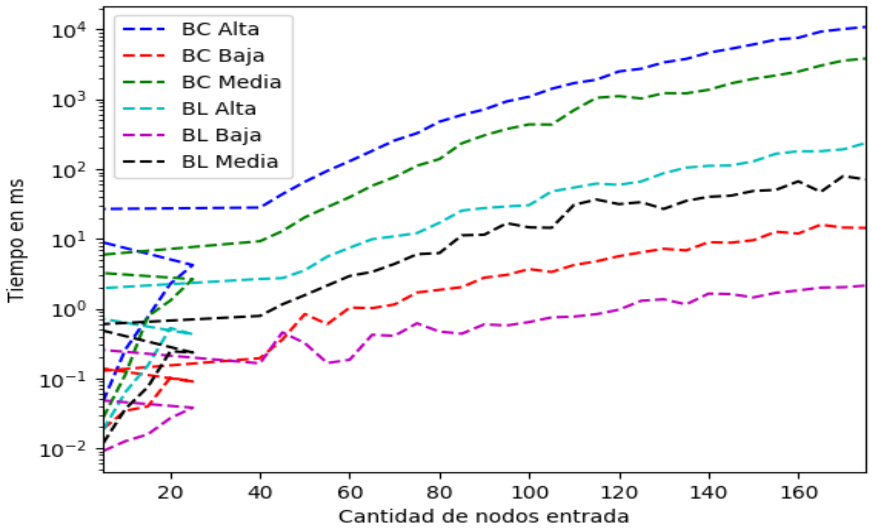
En un principio esperábamos el resultado contrario, partiendo de la hipótesis, de como en cada instancia ahora tiene más vecinos. Es menos probable que sea la solución actual sea un máximo local. Por lo que costarían más iteraciones encontrar una instancia que lo sea.

Sin embargo al ver que el comportamiento era justamente lo contrario, buscando una razón. Concluimos en que se debía a que, el aumento del conjunto de vecinos jugaba el papel contrario a lo que pensábamos. Al evaluar más casos por iteración, significa que al mismo tiempo, estás descartando más casos por iteración. Lo que concluye en una limpieza más rapida de las soluciones no óptimas. Y nos lleva a encontrar un pozo en la solución de manera más rapida.

Experimento Complejidad Temporal

En los siguientes gráficos ploteamos el tiempo de cómputo total promedio para los algoritmos de búsqueda local y cuadrática, para las tres densidades con las que venimos trabajando

En este caso elegimos utilizar los resultados promedios principalmente porque el tiempo de computo suele estar más atado a lo que son las magnitudes de cantidade de nodos y aristas. Que a la distribución del grafo. En esto último sin embargo obviamente hay una relación, ya que dependiendo de la distribución se puede iterar más o menos veces. Pero Com vimos en los gráficos anteriores, no es un factor que te cambia la complejidad asintótica del algoritmo en general.



Tiempos de BL (Búsqueda Local Lineal)
vs
Tiempos BC (Búsqueda Local Cuadrática)
Con diferentes densidades

A partir de los datos obtenidos, se puede ver efectivamente, que el aumento del tamaño del conjunto de vecinos trajo consigo una fuerte diferencia en el tiempo de cómputo. Exceptuando las lineas de complejidad para grafos con densidad baja, que de forma entendible se muestran como los más rápidas en ejecución. En los algoritmos de búsqueda local lineal vs búsqueda local cuadrática con densidades ya más elevadas, se ve como no pasa desapercibido este nuevo elemento a tener en cuenta. Por más que en la sección anterior vimos que la cantidad de iteraciones del algoritmo lineal suele ser mayor, para casos promedio finalmente el tiempo de cómputo por cada una marca la diferencia.

5 Meta-Heurística GRASP

La meta-heurística que fue implementada para resolver el problema de hallar la clique con la frontera de aristas más grande posible, fue hecha con GRASP. GRASP (Greedy Randomized Adaptive Search Procedures) consiste en un algoritmo que construye soluciones por medio de un algoritmo goloso, con la excepción de que en cada iteración, en vez de deterministicamente hacer una elección de la mejor decisión a tomar, en base a algun criterio ya definido, GRASP crea una lista de candidatos a partir de los cuales elige aleatoreamente con cual de ellos avanzar en la construcción de su solución.

Este conjunto de candidatos se construye utilizando el mismo criterio que la heurística golosa. Analiza que vertices del grafo se pueden agregar a la clique actual, de tal manera que maximizen la frontera más que cualquier otro vertice.

Cuando la heurística llega a una potencial solución, luego intenta mejorarla aún más con el uso de una busqueda local.

Todo este procedimiento se repite una serie de veces que puede ser definida estatica o dinamicamente. Por ello nos referimos a lo siguiente:

- Corte estático: se le asigna una cantidad definida de iteraciones al ciclo GRASP.
- Corte dinámico: se le asigna una cantidad máxima de iteraciones del ciclo GRASP en las cuales esta permitido no devolver una mejor solución. Luego de k ciclos sin mejora en la solución, entonces la meta-heurística se detiene.

Para definir los candidatos a solución dentro de la construcción de la clique, se utiliza un parámetro alfa el cual es utilizado en la siguiente comparación para determinar que vertices son candidatos y cuales no. Consideremos la función f, que por cada $v \in V$ retorna en f(v) la cantidad de aristas que aportan a la frontera de la clique en el caso de unirsele. Si un vértice cumple la siguiente condición entonces ingresa en la lista de candidatos, donde $F = \max_{v_i \in V} f(v_i)$ y $f = \min_{v_i \in V} f(v_i)$.

$$ListaCandidatosR = \{v \in V | F - f(v) \leq \alpha * (F - f)\}$$

5.1 Heuristica Golosa

La heuristica golosa aplicada que define el criterio con el cual se arma la lista de candidatos, es similar (casi identica) a la previamente vista *MFCGM*.

- Para cada iteración se cuenta con un vector de candidatos y un vector de vecinos. El primero de estos es un listado de los vertices que pueden agregarse a la clique y que esta siga siendo efectivamente una clique. Vecinos, por otro lado, es un vector que por cada vertice del grafo que no este en la clique, almacena cuantas aristas (que no incidan en la clique) estos vertices tienen.
- Usando la condicion en base al α previamente mencionada, segun el valor en el vector *vecinos* de cada vertice se define cuales de los candidatos pueden entrar a la lista reducida. Entre estos se elige aleatoreamente uno, llamado *elElegido*, el cual se agrega entonces a la clique.
- Finalmente, se redefinen los valores del vector *vecinos* ahora que este nuevo nodo se agrego a la clique. Tambien se limpia el vector de candidatos y se lo vuelve a llenar con los vertices que aun puedan formar una clique con la solución actual.

La obvia diferencia con *MFCGM* es que el vértice agregado en cada iteración termina siendo elegido al azar, pero decimos que ambas heuristicas golosas son similares ya que utilizan el mismo criterio de agregar nodos en función de "la mayor cantidad de aristas que aportan a la clique", para definir como construir la solución.

5.2 PseudoCódigo

Este pseudocódigo corresponde a la implementación de la busqueda de la clique con mayor frontera, utilizando el corte dinámico para definir cuantos ciclos GRASP se correran antes de terminar la ejecución. La alternativa sería no resetear el contador de ciclos actuales cada vez que se encuentra una solución de mejor resultado.

Algorithm 6 CliqueMaxFrontGRASP

```
procedure CliqueMaxFrontGRASP(Graph grafo, int α , int iterGrasp )
    solActual ← clique vacía sin frontera
    ciclos ← 0
    while ciclos < iterGrasp do
        vecinos ← vector de longitud grafo.nodos
        CalcularGrados(vecinos)
        clique ← clique vacía sin frontera
        listaReducidaCandidatos ← vector vacio
        candidatos ← vector vacio
        grafo.AgregarSiHaceClique(clique, candidatos)      ▷ Agrega a candidatos si son adyacentes a toda la clique
        while candidatos.length > 0 do
            listaReducidaCandidatos.clear()
            max ← DevolveMaxCandidato(vecinos, candidatos)
            min ← DevolveMinCandidato(vecinos, candidatos)
            for cand ∈ candidatos do
                if EsCandidato(vecinos, cand, max, min, α) then
                    listaReducidaCandidatos.pushBack(cand)
            if listaReducidaCandidatos.length = 0 then
                Break;
            elElegido ← random(0,listaReducidaCandidatos.length)
            grafo.AgregarFrontera(clique, listaReducidaCandidatos[elElegido])      ▷ Actualiza la clique
            grafo.ContarVecinos(clique, vecinos) ▷ Agrega a vecinos cuantas aristas aportaría cada nodo a la frontera
            candidatos.clear()
            grafo.AgregarSiHaceClique(clique, candidatos)
        clique ← BusquedaLocalLineal(grafo,clique)
        ciclos ← +1
        if clique.frontera > solActual.frontera then
            solActual ← clique
            ciclos ← 0
    return solActual
```

5.3 Experimentación

5.3.1 Cantidad Iteraciones

En esta experimentación se propuso ver cómo son los resultados obtenidos por la metaheurística GRASP a medida que la cantidad de iteraciones se incrementa. Para ello, se evaluó con grupos de grafos creados azarosamente (donde para cada dos nodos se definió si hay una arista utilizando la funcion *rand()*), cuyos tamaños fueron incrementando desde 20 a 500 vertices, con una granularidad de 20. El parámetro α se fijo en 0.5. Las iteraciones se fueron incrementando de a 10 hasta llegar a 200. Todo esto se realizó con la implementación de "corte dinámico", por lo que la cantidad de iteraciones de cada ejecución fue mayor o igual a la que se pasó por parámetro. Esta relación entre iteraciones permitidas e iteraciones realizadas también sera evaluada a continuación.

A partir de los resultados obtenidos se espera poder ver cómo mejoran los resultados de la metaheurística en función de la cantidad de iteraciones. También se busca observar si existe algún valor óptimo de iteraciones a partir del cual incrementarlo deje de ser significativamente mejor. Para ello, se obtuvo el siguiente *Heatmap*, para el cual el eje x representa la cantidad de Iteraciones Permitidas, el eje y la cantidad de nodos, y el color el resultado de la heuristica estandarizado.

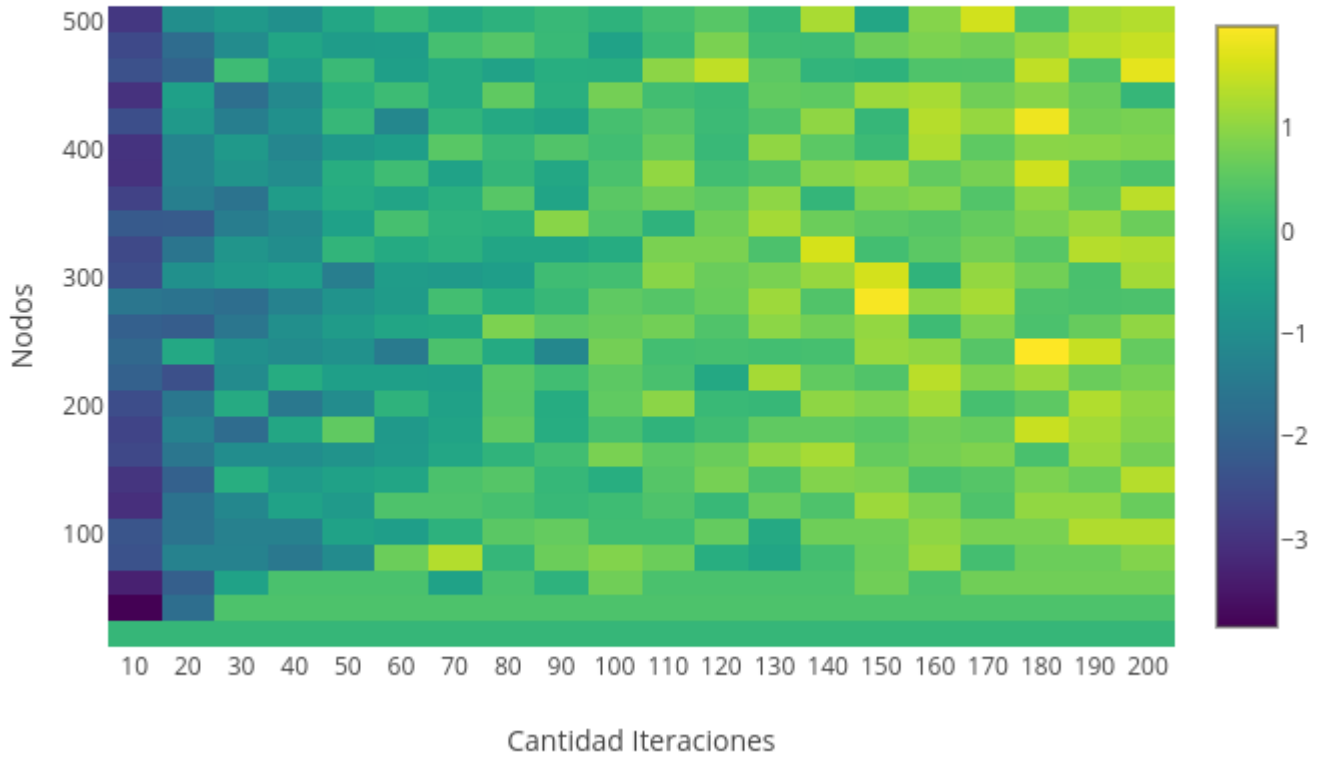


Figura 5.1 - Iteraciones vs Resultados estandarizados

Como GRASP utiliza la elección azarosa para construir sus soluciones, es posible (si no probable), que existan situaciones donde entre dos ejecuciones de la heurística, una teniendo más iteraciones permitidas que la otra, tenga un resultado menor. Esto mismo se puede ver en los resultados obtenidos.

Sin embargo en lineas generales se puede notar la tendencia de que, tal como se planteó previamente, la calidad de las soluciones obtenidas definitivamente aumenta en función de la cantidad de iteraciones permitidas. Pero se puede apreciar que a medida que los grafos van aumentando de tamaño, la cantidad de iteraciones a partir de la cual se obtiene un resultado promedio también va en aumento. Esto parecería indicar que la cantidad necesaria de iteraciones permitidas para obtener resultados más cercanos a la solución óptima, es proporcional al tamaño del grafo. A partir

de ésta observación no podemos llegar a la conclusión de que existe una cantidad de iteraciones a partir de la cual aumentar la misma no provee mejoras significativas.

5.3.2 Parámetro α óptimo

En este caso se fijó la cantidad de iteraciones permitidas en 150 y se realizaron mediciones de los resultados obtenidos por el GRASP para grafos de tamaños entre 20 y 500 nodos, con una granularidad de 20. El propósito de la experimentación es analizar cómo las soluciones devueltas varían en función del parámetro α, el cual se evaluó desde 0 a 1 con una granularidad de 0.05. Al igual que en el caso anterior, la graduación de los colores del Heatmap indican los resultados estandarizados.

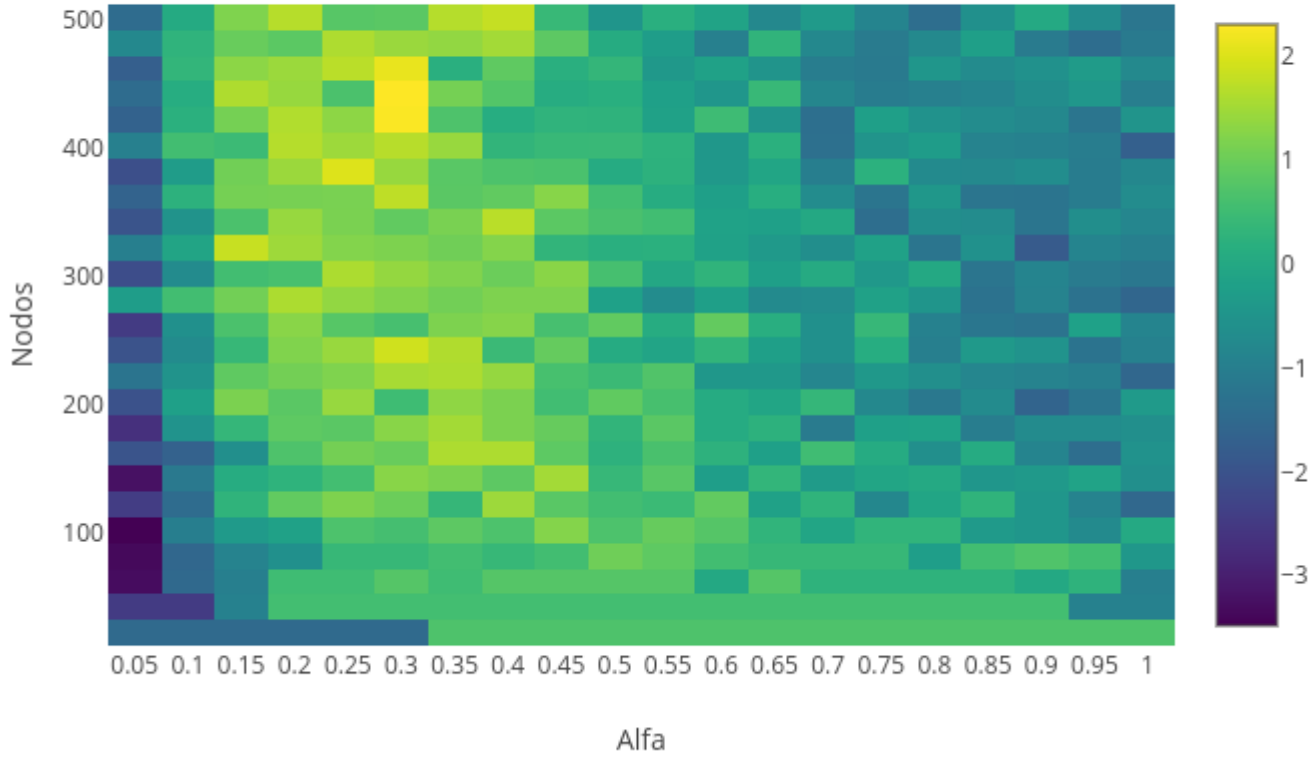


Figura 5.2 - Parametro Alfa vs Resultados estandarizados

Los resultados obtenidos parecen indicar que existe un valor de alfa para el cual los resultados obtenidos alcanzan su punto más alto. En los datos del heatmap, éste valor rondaría entre 0.25 y 0.30. El mismo se utilizará luego en la experimentación final para comparar todas las soluciones planteadas al problema de la clique de máxima frontera.

5.3.3 Iteraciones Permitidas vs Realizadas

Para ésta experimentación se intentó establecer alguna correlación entre la cantidad de iteraciones que se le permiten a la implementación de GRASP con corte dinámico, y la cantidad que realmente realiza.

La razón detrás de esto es que al tener una cantidad indefinida de ciclos de ejecución, ciertas comparaciones con otras heurísticas son difíciles de realizar (como el tiempo de ejecución por ejemplo). Por ende se intentó establecer una manera de inferir cuantas iteraciones se van a ejecutar a partir de la cantidad de iteraciones permitidas.

Para ello se hicieron mediciones de la cantidad de iteraciones realizadas en tandas de 30 grafos generados al azar tal como se explicó antes, de 500 nodos y 250. La cantidad de iteraciones permitida se evaluó desde 10 hasta 200 con una granularidad de 10.

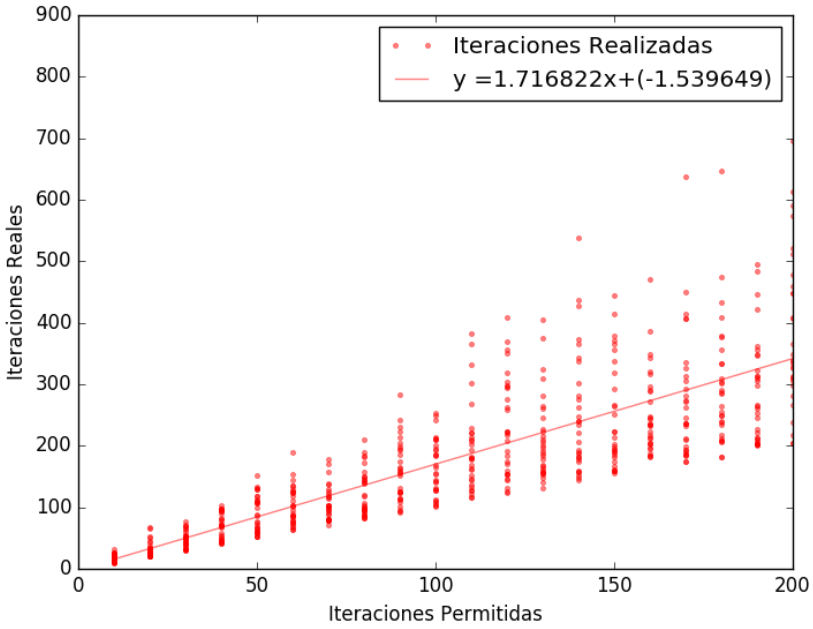


Figura 5.3 - Iteraciones Permitidas vs Realizadas (500 nodos)

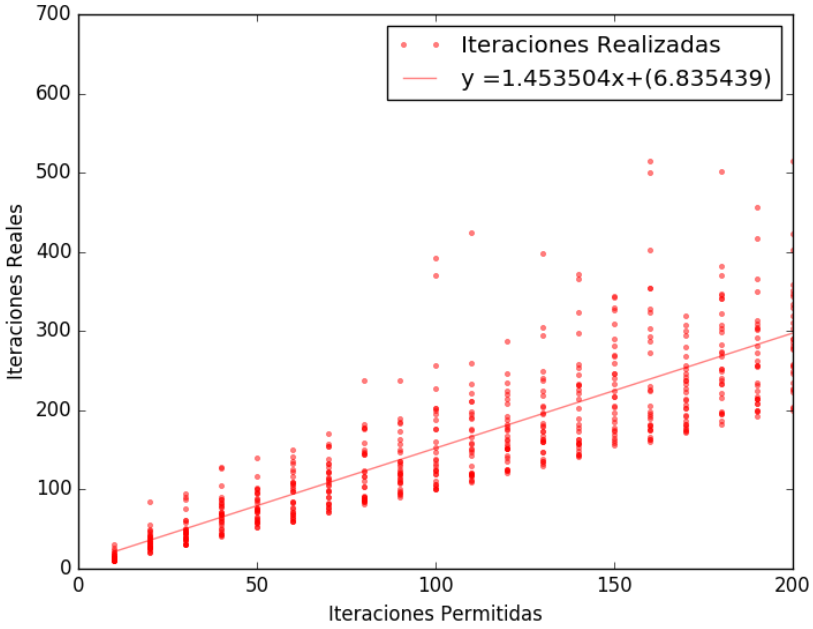


Figura 5.3 - Iteraciones Permitidas vs Realizadas (250 nodos)

A partir de los datos obtenidos se puede inferir que al menos tomando Iteraciones Permitidas hasta 200, existe una correlacion entre ambas variables, de orden lineal. Se definió entonces el valor de una constante $k = 2$, tal que $IteracionesRealizadas = k * IteracionesPermitidas$. Sin embargo, se puede observar como a medida que aumenta la variable en el eje x, las iteraciones reales se van dispersando más a lo largo del eje y, por lo que es probable que de evaluar ejecuciones de GRASP con aun más cantidad de ciclos ésta correlación no se mantenga.

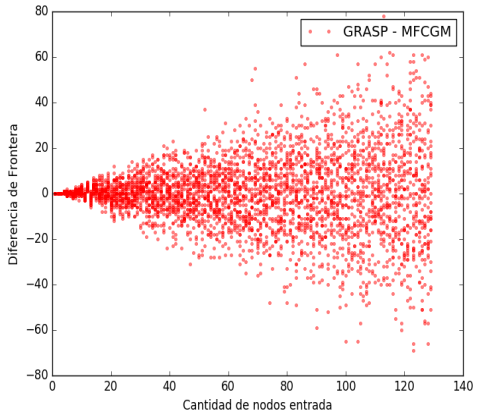
5.3.4 Experimentación general

En esta serie de experimentaciones se intentó registrar y analizar las diferencias entre los resultados obtenidos entre la ejecución de GRASP y el resto de las heurísticas previamente analizadas, sobre una serie de grafos idéntica para

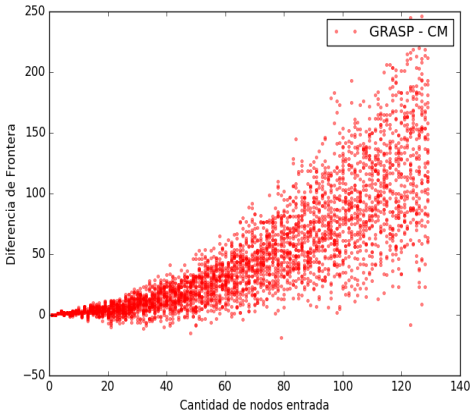
todos las implementaciones usadas (aquellas utilizadas en las experimentaciones de las heurísticas previas). Éstos se generaron al azar con la misma semilla, con una densidad media (0.5).

GRASP - Heurísticas golosas en Grafos Random

En este caso se utilizaron las dos heurísticas golosas propuestas para la solución del problema de Clique de Máxima Frontera. Estas eran la "maxima frontera cercana al grado máximo" (MFCGM) y "clique maximal" (CM). Podemos suponer por los resultados ya obtenidos en las otras experimentaciones que GRASP obtendrá resultados mucho mejores que CM, por lo que lo más interesante a analizar en este caso es cómo se comporta con respecto a MFCGM.



Diferencias Entre Heurística MFCGM y GRASP

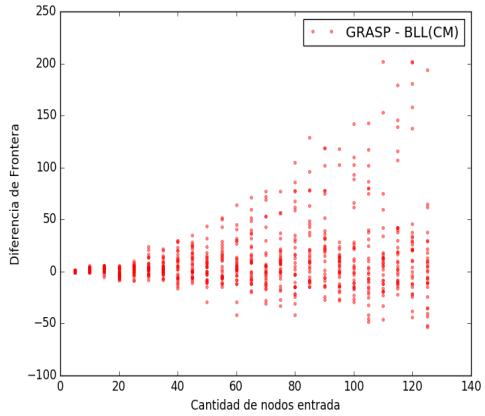


Diferencias Entre Heurística CM y GRASP

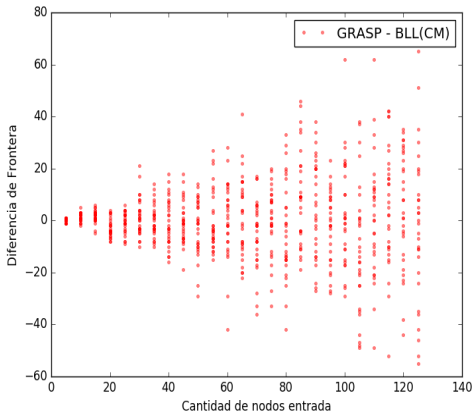
Como era de esperar, las diferencias entre GRASP y CM en casi todos los casos fueron positivas, por lo que los resultados de GRASP fueron más cercanos a la solución óptima. En el caso de MFCGM, esto no resultó de tal manera. Por más que la mayor diferencia registrada esta en favor de GRASP, los datos se encuentran casi simétricamente distribuidos por el eje y=0, lo que parecería indicar que hay tantos casos donde una de las heurísticas es mejor que la otra, como cuando no lo es. En el caso analizado, de los 3900 grafos tomados en cuenta, la sumatoria total de los resultados obtenidos por ambas heurísticas difirió en tan solo 1400 aristas a favor de GRASP. Sin embargo, esta diferencia poco significativa (1/3 de arista), podemos suponer que no vale la pena considerando el costo temporal que GRASP conlleva (al tener una complejidad indefinida).

GRASP - Búsquedas Locales en Grafos Random

Para este caso se utilizaron las dos búsquedas locales desarrolladas previamente. Tanto para la lineal como la cuadrática, a ambas se le dieron instancias de los mismos grafos anteriores con una solución ya resuelta por la heurística CM. Las búsquedas locales tuvieron que mejorar estos resultados iniciales, y luego se compararon con las mismas soluciones devueltas por GRASP analizadas en la sección anterior. Se esperó poder observar si alguna de las dos búsquedas podía mejorar las soluciones de la heurística CM lo suficiente como para superar a GRASP.



Diferencias Entre Búsqueda Lineal sobre CM y GRASP



Diferencias Entre Búsqueda Cuadrática sobre CM y GRASP

En los datos obtenidos se puede ver con claridad cómo en el caso de la búsqueda lineal los resultados que ésta heurística genera no logran superar los de GRASP. Las diferencias por lo general se mantienen más por encima del 0, y también hay gran cantidad de outliers por el lado de GRASP, que superan a la búsqueda lineal por hasta 200.

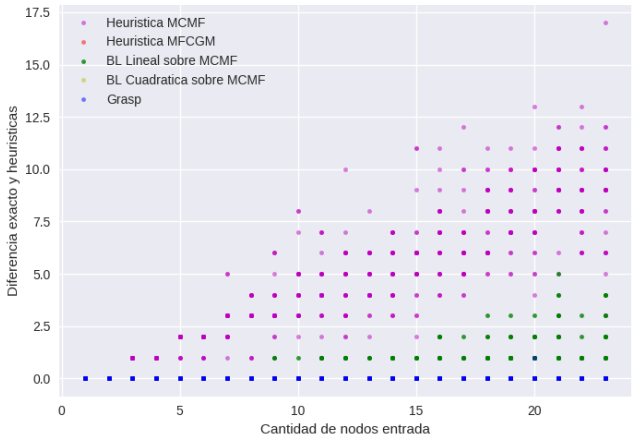
Sin embargo, tomando a consideración la búsqueda cuadrática, no se puede llegar a la misma conclusión tan fácilmente. Los datos se encuentran bastante distribuidos a lo largo del eje y, sin posibilidad de distinguir con facilidad cuál de las dos heurísticas supera a la otra. Para tratar de llegar a alguna conclusión se analizó la suma de los resultados obtenidos por las ejecuciones en todo el conjunto de grafos, el cual alcanzaba los 750 (se evaluó desde 5 a 125 nodos, con una granularidad de 5, de a tandas de 30 grafos). La diferencia entre estas sumas no superó los 100. Por lo tanto podemos inferir que al menos hasta los grafos de dicho tamaño, las diferencias entre el rendimiento de ambas heurísticas no es significativa.

6 Resultados finales

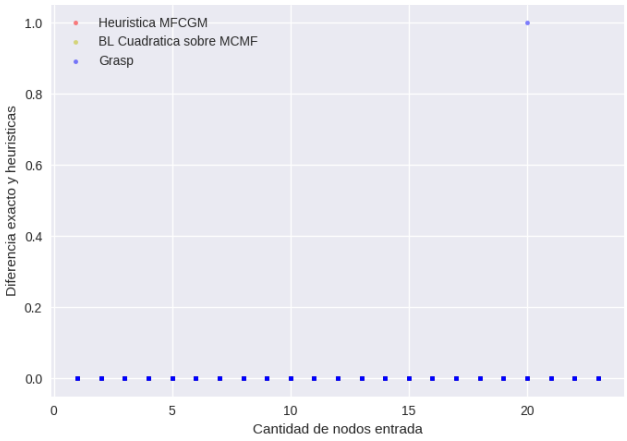
Se realizó un nuevo set de experimentos utilizando instancias generadas al azar diferentes a las usadas anteriormente con el objetivo de realizar una experimentación final entre todas las heurísticas implementadas. Como parámetros de grasp se utilizó un α de 0.21 y 200 iteraciones fijas. Ésta cantidad de iteraciones no nos permitió abarcar una gran cantidad de nodos pues grasp para una única instancia de $n = 200$ tiene un tiempo de cómputo de aproximadamente 60 segundos. También debemos aclarar que sólo se aplicó búsqueda local sobre la heurística *MCMF*, pues fue demostrado previamente que ésta no tiene ningún efecto sobre *MFCGM*.

6.1 Comparación Exacto vs Heurísticas

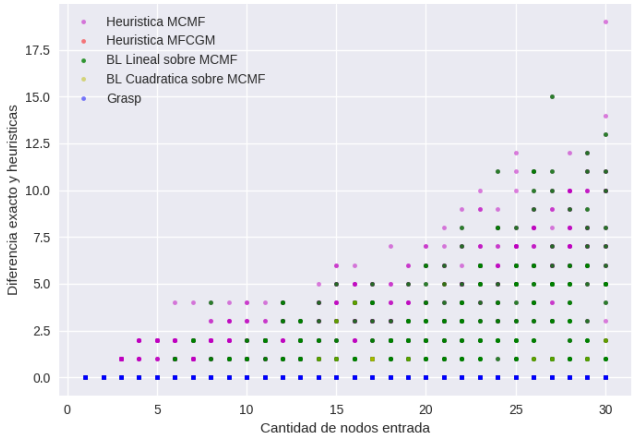
Se separaron instancias de tamaños pequeños para realizar una comparación con el algoritmo exacto, en las que se realizaron para cada cantidad de nodos n entre 1 y 23, 30 o 35 (densidades alta, media y baja respectivamente), 30 instancias diferentes. Se graficó la diferencia entre el resultado del algoritmo exacto y cada heurística:



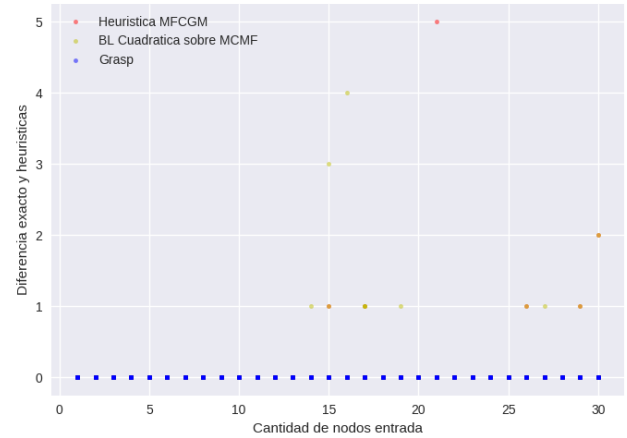
(a) Diferencia exacto y heurísticas densidad 0.8



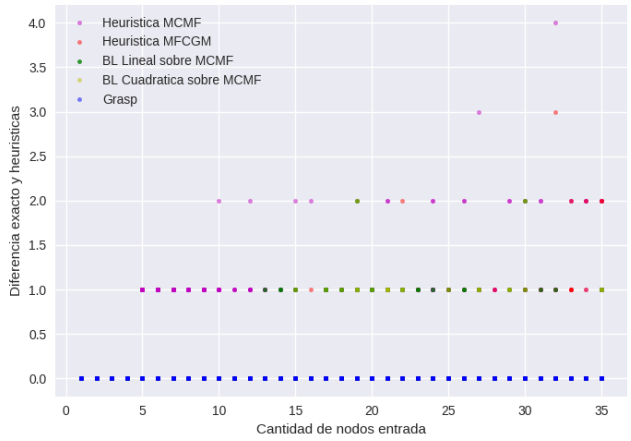
(b) Diferencia exacto y sólo 3 heurísticas densidad 0.8



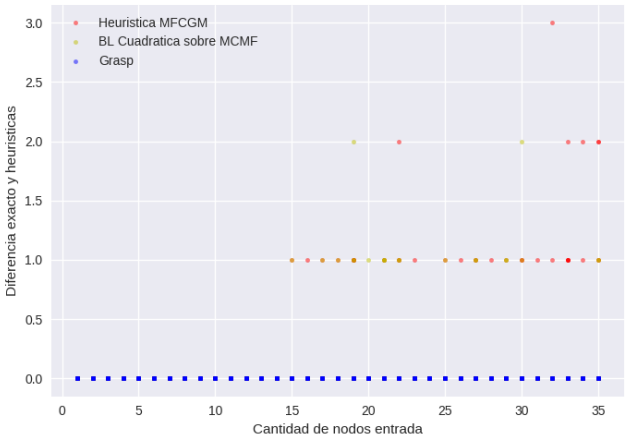
(a) Diferencia exacto y heurísticas densidad 0.5



(b) Diferencia exacto y sólo 3 heurísticas densidad 0.5



(a) Diferencia exacto y heurísticas densidad 0.1



(b) Diferencia exacto y sólo 3 heurísticas densidad 0.1

Se decidió graficar siempre a la derecha los mismos gráficos pero únicamente con 3 heurísticas (GRASP, Golosa MFCGM y Búsqueda local cuadrática sobre MCMF) pues éstas son las que devuelven resultados más cerca del exacto según resultados de experimentaciones anteriores.

Observando las figuras vemos varias cosas:

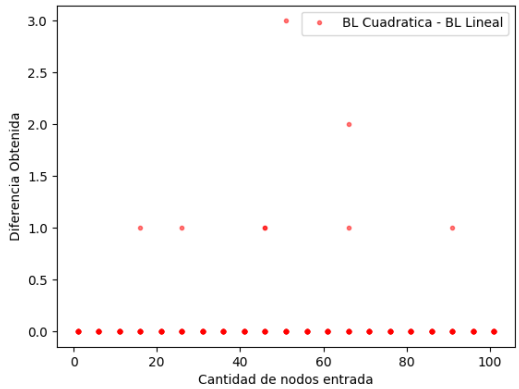
- Efectivamente la heurística *MCMF* y la Búsqueda lineal sobre ésta son las que devuelven resultados más lejanos de la solución óptima.
- La metaheurística GRASP funciona muy bien en general cuando tenemos cantidades de nodos pequeñas (sin importar la densidad de aristas). Analizando los gráficos parecería que con grafos al azar, por lo menos para éstas cantidades de nodos es la que mejor funciona.
- Las heurísticas *MFCGM* y Búsqueda Local cuadrática sobre *MCMF* devuelven cada vez más veces soluciones distintas de la óptima cuando la densidad es más pequeña. Ésto, si bien es una conclusión que comprende a sólo 2 de las heurísticas, es algo que no vimos en experimentaciones previas puntualmente en el caso de Golosos, y parcialmente en el caso de la búsqueda local cuadrática sobre *MCMF*

6.2 Comparando Resultados

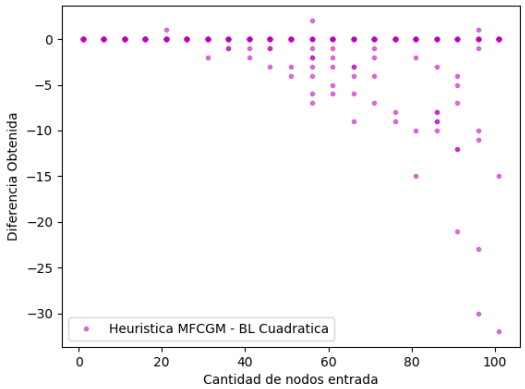
En esta sección comparamos nuevamente todas las heurísticas, con un nuevo lote de grafos con densidades 0.1, 0.5 y 0.8.

Para representar claramente las diferencias entre los resultados devueltos por cada heurística, lo que hicimos fue plantear gráficos que representen la diferencia entre las soluciones de los distintos algoritmos. En consecuencia, tendremos demasiados gráficos para generar en caso de querer comparar todos los pares de heurísticas. Para resolver este problema pensamos simplemente en ir comparando desde los algoritmos menos eficientes, y cuando se encuentre una heurística que acote completamente de forma superior a otra, simplemente continuamos las comparaciones con la de mejores respuestas.

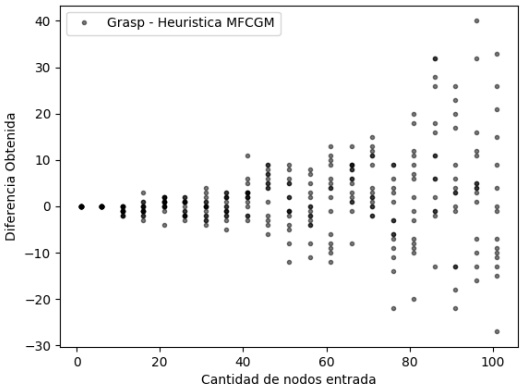
Comparando todas las Heuristicas Con Densidad 0.1



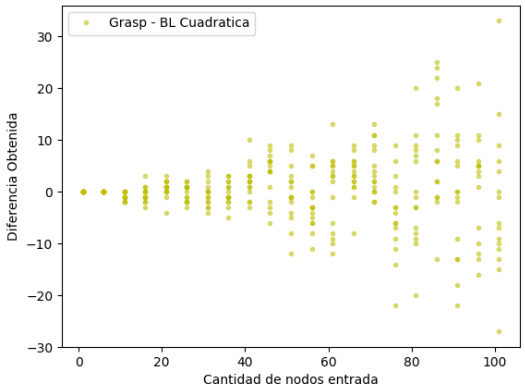
Diferencia entre BL Cuadratica y BL Lineal, aplicadas con semilla MCMF



Diferencia entre Heuristica golosa MFCGM y BL Cuadrática, con semilla MCMF

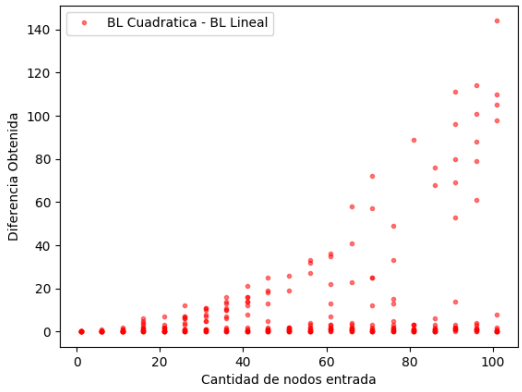


Diferencia entre Grasp y Heuristica MFCGM

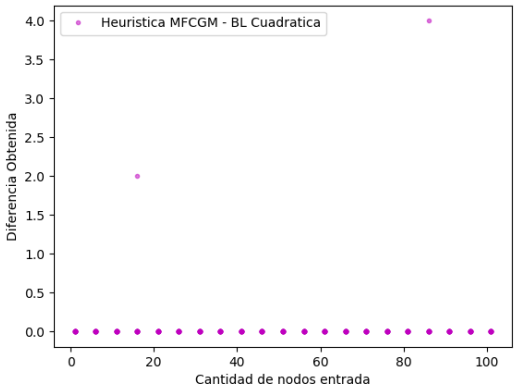


Diferencia entre Grasp y BL Cuadratica

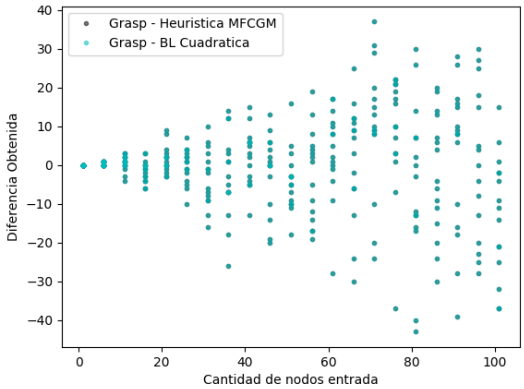
Comparando todas las Heuristicas Con Densidad 0.5



(a) Diferencia entre BL Cuadratica y BL Lineal, aplicadas con semilla MCMF

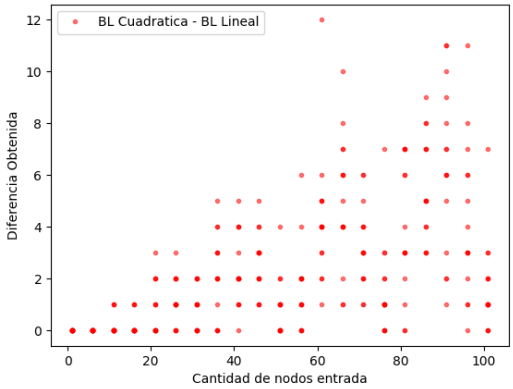


(b) Diferencia entre Heuristica golosa MFCGM y BL Cuadrática, con semilla MCMF

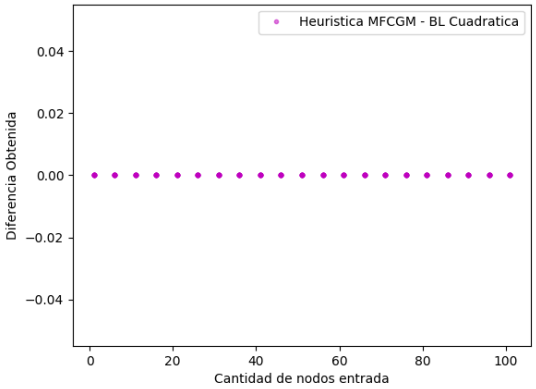


(a) Diferencia entre Grasp, Heuristica MFCGM y BL Cuadratica

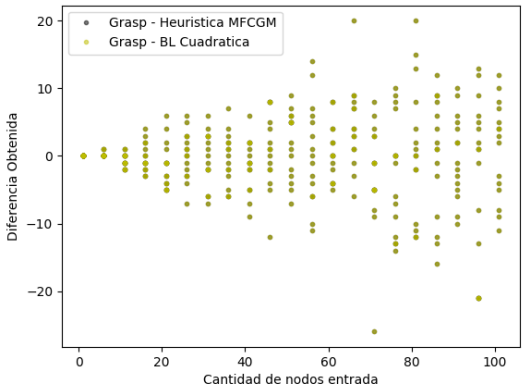
Comparando todas las Heuristicas Con Densidad 0.8



(a) Diferencia entre BL Cuadratica y BL Lineal, aplicadas con semilla MCMF



(b) Diferencia entre Heuristica golosa MFCGM y BL Cuadrática, con semilla MCMF



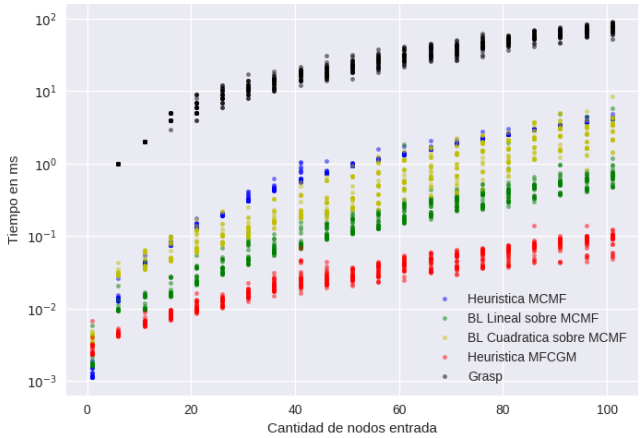
(a) Diferencia entre Grasp, Heuristica MFCGM y BL Cuadratica

La conclusión final de ésta experimentación es que en verdad a lo largo del trabajo práctico no se produjo overfitting, pues obtuvimos en lineas generales resultados muy similares a los experimentos ya analizados, con obviamente ciertas variantes surgidas de cambiar el lote de grafos.

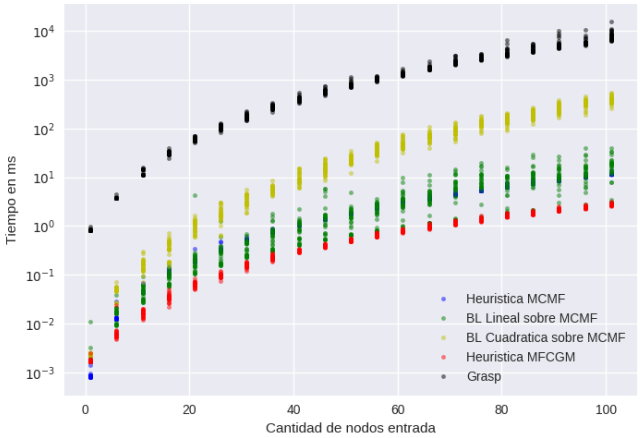
No hay muchas más conclusiones que hacer pues, como ya dijimos, los resultados de la experimentación son similares a los de secciones anteriores.

6.3 Análisis de tiempos

Aquí comparamos los tiempos de cómputo de las heurísticas según la densidad de aristas. Omitimos el análisis de tiempo del algoritmo exacto pues por secciones anteriores ya sabemos que éste es exponencial y superará ampliamente el tiempo de cómputo de todas las heurísticas. Los grafos utilizados son los mismos que en la **sección 6.2**.



(a) Tiempos de cómputo densidad aristas 0.1



(b) Tiempos de cómputo densidad aristas 0.5

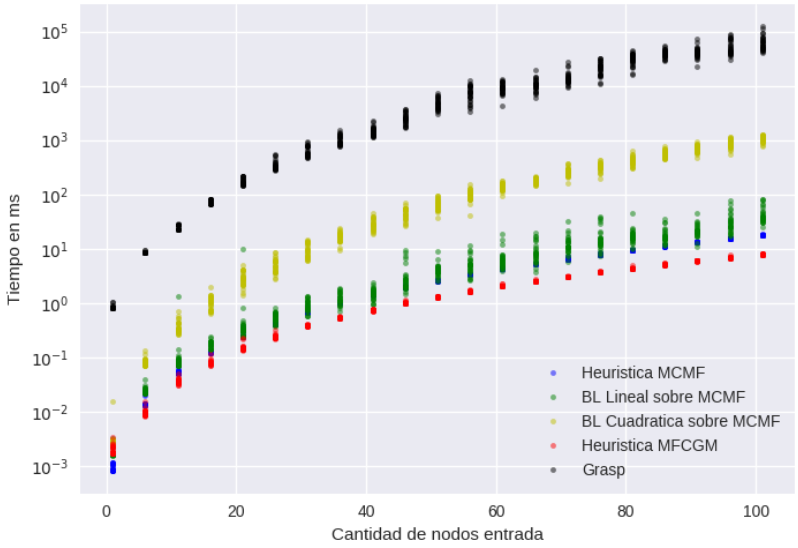


Figure 33: Tiempos de cómputo densidad aristas 0.8

Como podemos ver en las **Figuras**, sin importar la cantidad de aristas la metaheurística *grasp* tiene tiempos de cómputo peores. Una posible hipótesis es que esto se debe a la alta cantidad de iteraciones utilizada para intentar maximizar las posibilidades de que el algoritmo devuelva la solución óptima en cada instancia. Por otro lado, los tiempos de cómputo de las heurísticas y la búsqueda local son correlativos con los experimentos realizados en las secciones previas cuando se analizó experimentalmente el tiempo de cómputo de las mismas.